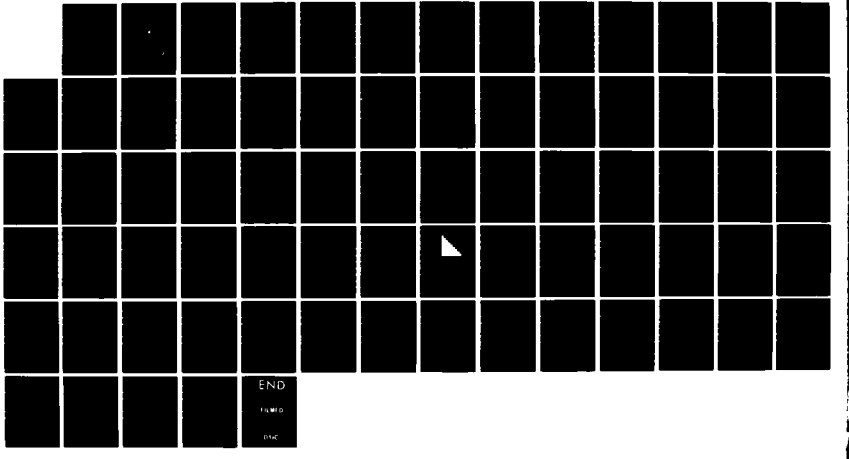
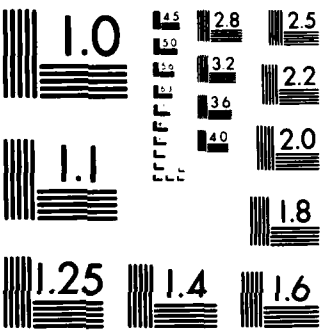
AD-A151 570 ROUTING A USCG BUOYTENDER TO SERVICE AIDS TO NAVIGATION: A CASE OF THE TRAVELING SALESMAN PROBLEM (U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA J M BECHTLE 1/1

UNCLASSIFIED SEP 84 F/G 15/5 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD-A151 570

(2)

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

ROUTING A USCG BUOYTENDER
TO SERVICE AIDS TO NAVIGATION:
A CASE OF THE TRAVELING SALESMAN PROBLEM

by

Jon Michael Bechtle

September 1984

Thesis Advisor: G. L. Lindsay

DTIC FILE COPY

DTIC
ELECTE
MAR 2 2 1985
S D
D

Approved for public release; distribution unlimited.

85 03 11 114

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A151 570	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Routing a USCG Buoytender to Service Aids to Navigation: A Case of the Traveling Salesman Problem	5. TYPE OF REPORT & PERIOD COVERED Master's Thesis September, 1984
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Jon Michael Bechtle	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE September, 1984
	13. NUMBER OF PAGES 71
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report)
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

traveling salesman, tour, k-optimal,

Accession For	
NTIS GRA&I	X
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A problem of routing a U.S. Coast Guard buoytender to service aids to navigation is formulated as a symmetric traveling salesman problem. A heuristic algorithm is developed which seeks the minimum distance tour which can be taken by the buoytender to visit the aids to navigation. A user's guide is provided.

The algorithm is programmed in Convergent Technologies FORTRAN for use on the Coast Guard Standard Terminal. Several problems are solved by the algorithm producing solutions that are optimal or nearly optimal.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Approved for public release; distribution unlimited.

Routing a USCG Buoytender to Service Aids to Navigation A Case of the Traveling Salesman Problem

by

Jon M. Bechtle
Lieutenant, United States Coast Guard
B.S., U.S. Coast Guard Academy, 1978

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN OPERATIONS RESEARCH

from the

NAVAL POSTGRADUATE SCHOOL
September 1984

Author: Jon M. Bechtle

Approved by: Glenn F. Lindsay
Thesis Advisor

R. Kevin Wood
Second Reader

Carl R. Jones
Chairman, Department of Operations Analysis

K. T. Marshall
Dean of Information and Policy Sciences

ABSTRACT

A problem of routing a U.S. Coast Guard buoytender to service aids to navigation is formulated as a symmetric traveling salesman problem. A heuristic algorithm is developed which seeks the minimum distance tour which can be taken by the buoytender to visit the aids to navigation. A user's guide is provided.

The algorithm is programmed in Convergent Technologies FORTRAN for use on the Coast Guard Standard Terminal. Several problems are solved by the algorithm producing solutions that are optimal or nearly optimal.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

One of the U.S. Coast Guard's missions is maintaining aids to navigation in the waters of the United States. These aids require periodic servicing to ensure they are on station and they are showing their proper watch characteristics. The responsibility of maintaining many of these aids falls on the Coast Guard's fleet of buoytenders. A buoytender in its area of responsibility may have as many as 200 aids which it is required to maintain.

The buoytender has scheduled ATON runs (Aids TO Navigation) and emergency outages it must handle during a fiscal year. An emergency outage occurs when an aid is reported by a mariner to be showing improper characteristics or is missing from its station. These outages are handled shortly after they are reported. Several times during the year the buoytender plans an ATON run to handle the periodic servicing of aids, to ensure that they maintain their proper watch characteristics.

Prior to a scheduled run the tender will receive SANDS forms from the unit's district office for each of the aids which are due for service or relief. SANDS is a database where information is kept on a district's navigational aids. The district's aids to navigation branch identifies aids which are due for service or relief and notifies the tender. The district provides a list of aids which need to be serviced during the scheduled run and of work needed on each aid. The tender then makes plans accordingly to schedule and complete the work.

The number of aids serviced on a particular scheduled ATON run may be as few as 5 (a day of local work) to as many as 70 (an extended four-week trip servicing the west coast

PREVIOUS PAGE IS BLANK

of Alaska from the Aleutian Islands to Point Hope). The maintenance of the floating aids and fixed structures entails the travel of the tender from aid to aid. The tender's Operations Officer usually plans the initial route to be travelled to service the aids due for routine maintenance and presents his proposal to the Commanding Officer, who may then modify this proposal.

There are several factors which go into the decision process to produce the desired route. Two of these factors are (1) which aids are scheduled for service, and (2) the distances between the aids. The desired route is usually the route of shortest distance which visits all the scheduled aids and returns the tender to her point of origin. This problem is a classic problem which operations analysts normally call 'the traveling salesman problem.' Present route selection methods are (1) traditional or previously followed routes and (2) routes created by Operations Department personnel sitting down with chart and rule and selecting a route.

It is the intent of this thesis to propose and implement a computer assisted approach as an alternative method for the buoytender route selection problem. The succeeding chapters of this paper contain a discussion of the traveling salesman problem and how it relates to the buoytender problem, a brief survey of possible solution methods to the problem, a description of the solution process of the selected method, and the results of the route selection program with conclusions.

II. NATURE OF THE PROBLEM

In this chapter the principles of the traveling salesman problem will be presented and it will be shown how the buoy-tender problem can be described as a traveling salesman problem. The discussion will include a general description, definitions about solution methods, and the solution requirements of the problem. Further, a description of the computing hardware on which the selected solution method is programmed is provided. This provides an idea of the capabilities of the hardware since the hardware has a bearing on the selection of the solution method.

A. DESCRIPTION OF THE TRAVELING SALESMAN PROBLEM

A brief description of the traveling salesman problem (or TSP) is as follows. A salesman has n cities he must visit. The salesman starts in one of the cities and must travel so that he enters and departs each of the other cities only once. Upon visiting the last city the salesman will return to the city from which he originally started his travels. The desired solution to the problem is the route the salesman can traverse which is of the shortest total distance possible.

It is usually assumed that a TSP can be associated with some n by n distance matrix. The elements of the matrix, d_{ij}, are distances from city i to city j, (i=1,...,n), (j=1,...,n), and where $d_{ii} = \infty$, i=1,...,n. The traveling salesman is not allowed to leave city i and to return to city i in his tour.

A TSP can be associated with a graph, also. A city which the salesman must visit is represented by a node in

this graph. There is an arc (i,j) in the graph with length d_{ij} for each node.

A distance matrix may be classified as sparse or dense. A sparse matrix will have a majority of its entries equal to infinity. The corresponding graph is sparse since it has few edges. A dense matrix is a matrix where a majority if not all its entries, with the exception of the d_{ii}'s, are less than infinity. The number of possible permutations of cities for a dense matrix is of order (n-1)!. Computation time and memory required for solution by total enumeration grows rapidly with the size of n due to the large number of possible solutions, each permutation of cities being a feasible tour and a possible solution.

The distance matrix used by the buoytender for route selection will be a dense matrix. With the large number of permutations or possible solutions to the problem, some means other than total enumeration must be used to solve the buoytender problem.

The traveling salesman problem may be broken into two classes, symmetric and asymmetric. In a symmetric TSP the $d_{ij} = d_{ji}$ for all $i \neq j$, while in an asymmetric TSP this need not be true. Further, a TSP may or may not be required to satisfy the triangle inequality. Consider three cities i, j, k. The triangle inequality states that

$$d_{ij} \leq d_{ik} + d_{kj} \qquad \text{for all } i \neq j \neq k \neq i.$$

The conditions which make a TSP asymmetric, the unequal distances between cities, may cause the problem to violate the triangle inequality.

The TSP, both symmetric and asymmetric, belongs to the NP Complete class of problems [Ref. 1,2]. This has one major implication with respect to the complexity of solving the TSP: no polynomial-time algorithms are known or seem likely to be devised for exact solution of the TSP. Only exponential-time exact algorithms are known.

There have been many algorithms proposed for the solution of the TSP, each algorithm showing some advantages and disadvantages to its method of attacking and solving the problem. Several of the published algorithms are tailored to a particular size of TSP. The TSP may be categorized by size as either small or large. A reference could not be found in the literature which would give a hard numerical value by which the size of a TSP could be judged, but a small TSP is generally considered to be less then 15 cities. A large ISP is often considered to be in the neighborhood of 45 or more cities. There are solution methods which solve the TSP exactly for small up to large problems. There are some methods which can solve some large problems exactly, but as a rule large problems are solved with heuristics or approximate methods.

The circumstances of the buoytender problem satisfy the conditions of a class of the traveling salesman problem. The buoytender problem can be characterized as a symmetric TSP, i.e., the distance traveled between any two aids is the same regardless of direction travelled by the tender. The size of buoytender problem will vary with the number of aids scheduled to be serviced on a specific ATON run; therefore the solution method for the buoytender problem will have to handle both small and large problems.

The triangle inequality as applied to the buoytender problem holds true, but how it holds needs some explanation. With all distances positive, the triangle inequality states that given three points A, B, and C, the distance from A to C must be less than or equal to the distance from A to B plus the distance from B to C (see Figure 2.1). In the buoytender's problem the straight-line distance from A to C may be less than (A to B) + (B to C), but some situations will require that A to C equal (A to B) + (B to C) (see Figure 2.1),

$$d_{AC} = d_{AB} + d_{BC} .$$

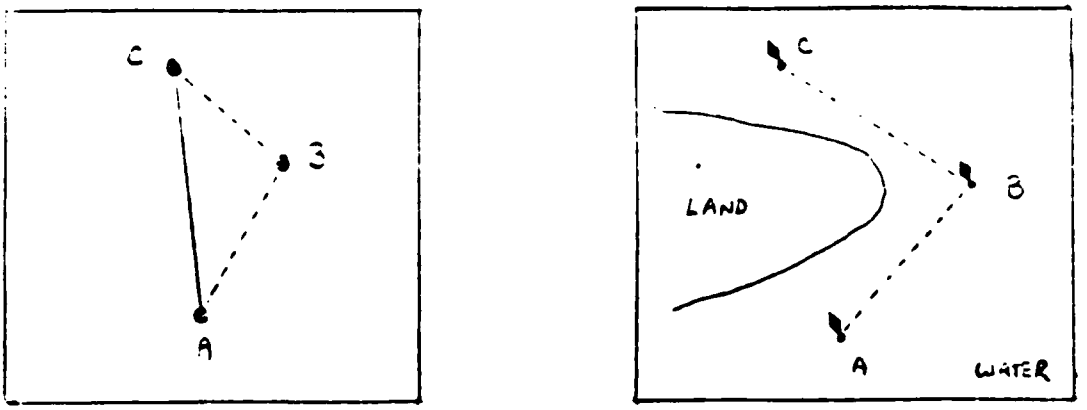


Figure 2.1 Triangle Inequality / Obstructed A to C Path

The buoytender may be unable to take the direct route from A to C due to shoal water or an obstruction. To get from A to C the tender will have to take the routes A to B and B to C. This is not an unusual situation for buoytenders since the aids being serviced often mark shoal water or obstructions.

B. DESCRIPTION OF THE SOLUTION REQUIREMENTS

There are many proposed solution methods to the TSP and they may be divided into two groups: exact methods and approximate or heuristic methods. An exact method or algorithm has the property that upon termination of the algorithm, the user will have the best possible or global optimum solution. Heuristics or approximate methods, on the other hand, terminate with a local optimum solution or the best solution found thus far. This local optimum solution may be the global optimum but usually there is no proof that the global optimum has been found. Heuristics or approximate methods solve by checking a subset of the numerous possible solutions and terminate when the subset has been searched for the best answer it contains, a specified time

limit for the run has been reached, or the memory capacity of the computing machine has been reached.

A further consideration in selecting a method to solve the TSP is that there should be very little interactive work for the user. The user interactive work necessary to operate the program should be understandable to a majority of the intended users, who in this case are the buoytender's Operations Officer or Navigator, and Quartermaster personnel. By requiring the interactive work to be minimal and simple in nature the possiblity of operator introduced error will be reduced. Further, if the interactive work is of a simple nature, the program will be useable by a majority of the intended users without extensive training.

Although it is desired to obtain the global optimum, it is not mandatory. The computer route selection method is to be a tool used in planning, and the exact solution is not critical to the operation of the buoytender. By using heuristics, computer time and memory may be reduced since a heuristic generally searches only a subset of all possible solutions. If a computer solution can be produced which is better than the traditional route or a route produced by hand calculation, then a goal has been met, that goal being reducing the total distance traveled by the tender thereby saving fuel and time. The computer solution is then a viable alternative tc be added to the present methods of route selection.

C. COMPUTING HARDWARE

The computer on which the selected solution method is programmed is the Coast Guard's C3 Standard Terminal. The reascn for selecting this computer is that it is presently being installed throughout the Coast Guard as a primary unit-level computer. Most intend users should have access to a C3 Terminal.

The C3 Standard Terminal is a standalone 16 bit microcomputer. The terminal comes in two basic configurations, the Integrated Work Station (IWS) and the Application Work Station (AWS). The IWS is a master or standalone unit equipped with a hard disk and 8 inch floppy mass storage system, a 300/1200 baud modem, a printer, and one of three RAM memory configurations - 256k, 384k, or 512k. The IWS may be programmmed in BASIC, FORTRAN, COBOL, Pascal, and Assembly. The AWS is more limited in its capabilities than the IWS. The AWS is limited to 256k RAM and must be networked to a IWS station to access the peripherals. With a program in RAM the AWS will operate as a standalone computer, but does not have the power or computational speed of the IWS.

The solution method selected for programming will have to perform within the limitations of the C3's memory. The C3's RAM memory will be a critical factor in selection of the solution method. Since the computer is a purchased standalone and not a leased system, CPU time should not be a critical factor. It is still desired, though, to keep CPU time from becoming excessive since the computer is also needed for other work. The computer could be allowed to run overnight (off duty hours) for large problems, and therefore a solution method requiring more than 12 hours to provide an acceptable solution would be excessive. A solution method which provides an answer in under an hour could be run during duty hours with very little impact on other users, particularly if it was run on a slave terminal.

In the next chapter we take a brief look at some suggested solution methods for the TSP and their applicability to the buoytender problem.

III. A BRIEF SURVEY OF SOLUTION METHODS FOR THE TSP

In this chapter a brief survey of the general solution methods for the TSP and some of their associated algorithms will be presented. The number of algorithms proposed for solving the TSP is extensive. To present all of the available algorithms or to discuss them in detail is worthy of a dissertation in itself. This survey of solution methods is made because a combination of a couple of the methods will be used to solve the buoytender problem. This survey will be brief in nature presenting the general solution methods and briefly describing some of the more popular of published algorithms which fall under these solution method headings.[1]

The solution methods may be classified under four general headings. These headings are Tour Building, Subtour Elimination, The One-Tree Formulation, and Tour Improvement. These headings describe how the traveling salesman problem is approached.

A. TOUR BUILDING

Tour Building solution methods to construct a tour using the n nodes and the available arcs in the problem. Tour Building contains both exact and approximate algorithms. Some of the approaches taken by Tour Building Algorithms to solve the TSP are dynamic programming, 'branch and bound' methods, and tour construction heuristics.

---

[1]Three excellent articles [Ref. 1,3,4] have been published which describe the TSP, discuss general solution methods, and review many of the algorithms which solve it. These three articles review the research conducted on the TSP from the mid 1950's up to 1983.

Held and Karp [Ref. 5] present a dynamic programming approach which is suitable for small problems and can be modified to an approximate method for larger problems. The dynamic programming approach rapidly consumes available memory as the size of the problem increases. Due to memory limitations, the algorithm Held and Karp presented is exact only up to 13 cities. They also present a dynamic programming method which approximates the solution for larger problems. The approximation is done by partitioning the large problem into several smaller problems or subproblems, each subproblem containing 13 or fewer nodes. Each partitioned subproblem is solved for its optimal tour resulting in a set of subtours for the original problem. The subtours are then broken and linked to each other to provide a tour for the original large problem.

Little et al. [Ref. 6] present a branch and bound method. The algorithm branches on whether a particular arc is included or excluded from the tour being constructed. This branching partitions the solution set containing all tours into subsets. Each arc in the total graph can be associated with some subset of tours, and the idea is to find the subset containing the optimal tour. For each subset of tours, a lower bound is computed for the tours within the subset. The tour is constructed as arcs with favorable lower bounds are included in the tour. The algorithm continues branching and computing lower bounds until a subset is found which contains one tour and this tour's distance is less than or equal to the lower bounds of the other subsets of tours. This tour will be the optimal solution to the TSP. The memory required for this technique may be extensive and computing time grows exponentially with the size of the problem.

Many tour construction heuristics have been proposed. Rosenkrantz, Stearns, and Lewis [Ref. 2] and Golden et al.

[Ref. 4] review several of these heuristics. The two most popular forms of these heuristics are the nearest neighbor algorithm and the insertion techniques.

The nearest neighbor algorithm operates exactly as its name implies. A starting node is chosen and its nearest neighboring node is selected as the next node in the tour being constructed. The algorithm then iteratively selects a node not yet in the tour which is nearest to the previously selected node to enter the tour. This selection continues until all nodes have been selected and then the last node selected is connected to the first node to complete the tour.

The insertion techniques include the nearest, farthest, cheapest, and arbitrary insertion algorithms. Insertion techniques begin with a starting node and select the nearest node to create a subtour. The algorithm iterates selecting a node which is nearest to or farthest from any other node in the subtour, or arbitrarily selects the next node to enter the tour. This selected node is then inserted into the subtour wherever it provides the least increase (or in the case of the farthest, the greatest increase) in distance in the new subtour. These heuristics generally provide a suboptimal solution to the TSP, but they have the advantage of being very simple to implement and have very fast solution times with minimal memory consumption.

B. SUBTOUR ELIMINATION

The Subtour Elimination solution method [Ref. 1] attacks the TSP by solving an n by n assignment problem with the added constraints that the final solution must contain a cycle and the cycle cannot be of size n-1 or less. The number of integer constraints necessary to eliminate all subtours or cycles of size n-1 or less is enormous.

Therefore, the initial solution is generally found using a 'relaxed' form of the assignment problem. The relaxed assignment problem omits the subtour constraints in its formulation.

Mathematically stated the relaxed assignment problem is

$$\text{Min} \quad \sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij} d_{ij}$$

$$\text{S.T.} \quad \sum_{j=1}^{n} x_{ij} = 1 \qquad i=1,2,\ldots,n$$

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad j=1,2,\ldots,n$$

$$\text{where} \quad x_{ij} = 0 \text{ or } x_{ij} = 1$$

The solution to the relaxed assignment problem provides the initial lower bound on the optimal tour length. If the optimal assignment solution is a tour then it is an optimal solution to the TSP. If the assignment solution is not a tour then there exist subtours which must be removed until a single tour exists. If a subtour exists with k arcs in the subtour, then there are k possible subproblems to which the problem may branch. These k subproblems each have an additional constraint, the constraint for subproblem i (i=1,...,k) being the exclusion of arc i from the problem to eliminate the subtour. Instead of branching into k subproblems the approach is to branch into two subproblems. An arc is selected from the subtour and the exclusion of this arc becomes a new constraint in one of the subproblems while inclusion of the arc in the tour becomes a new constraint for the alternate subproblem. The modified assignment problem is solved again for each of these new constraints and provides the lower bound for its respective branch. In this way only the necessary subtour elimination constraints are added as needed to the assignment problem, rather than attempting to add all possible subtour constraints to the initial assignment problem. Ideally the branch solution subset with the lowest bound is then selected to be tested

to see if it is a tour, and, if it is not, the problem branches again on the lowest bound. This branching continues until all subtours have been eliminated and a tour is found which has a distance less than or equal to the lower bounds of the other solution subsets.

There are two basic methods for searching the tree created by the branch and bound process. One method is a breadth-first search. As the program is branching and computing bounds, a tree is created with leaves or terminal nodes. A breadth-first search requires storage for the solutions of each leaf and a search through these leaves for the next subproblem on which to branch. This requires extensive, usually exponential, storage. Branching at the lowest bound requires a breadth-first search. The second method for searching the tree is a depth-first search. The depth-first search is a more localized approach to handling the search. The depth-first search need only store the immediate solution while searching a branch. If during the search it is found that no further branching is possible from the current node, the process must return to a previous leaf to continue the search for the optimal tour. The depth-first search backtracks and recreates the previous solution at that leaf. The depth-first search does not require exponential storage like the breadth-first search, but it usually requires more computation time.

The subtour elimination method is described as being exact. The subtour elimination method usually requires less computation time than the branch and bound approach of Little et al. since the bounds obtained with the assignment relaxation are tighter.

C. THE ONE-TREE FORMULATION

The one-tree formulation was introduced and developed by Held and Karp [Ref. 7,8]. The general method for the solution of the TSP begins with the construction of a minimum-weight 'one-tree.' If the one-tree is not a tour then an integer linear program may be used to obtain the final solution.

To assist in the description of this solution method several terms need defining. First a tree is defined as a connected graph without cycles. A minimum-weight spanning tree is a tree with minimum total weight on its edges. A one-tree is a tree to which one edge has been added yielding exactly one cycle. A minimum-weight one-tree is a one-tree with minimum total weight on its edges. Finding a minimum-weight one-tree is a relaxed version of a TSP since the solution to a TSP is a minimum-weight one-tree having every vertex of degree two.

In the solution method proposed by Held and Karp [Ref. 7] the algorithm begins by constructing a minimum-weight one-tree. A minimum-weight one-tree can be can be found by first constructing a minimum-weight spanning tree through nodes 2 to n and then adding to the graph the two arcs of least weight from node 1 [Ref. 9]. A minimum-weight one-tree may also be constructed by constructing a minimum-weight spanning tree through nodes 1 to n and then add to the graph the next minimum-weight edge not yet used from the distance matrix. If the one-tree constructed is a tour (i.e. the one-tree's vertices are all of degree two) then the TSP is solved. Otherwise, the one-tree must be converted into a tour.

Integer linear programming is used to obtain an optimal tour if the one-tree formulation does not satisfy a tour. Held and Karp introduce the concept of a 'gap' function and

then use a special integer linear program to minimize this function. In essence there is a transformation on the one-tree variables which allows the integer linear program to search for the optimal tour. Held and Karp suggested two methods for optimizing the integer linear program, an ascent method and a branch and bound method with the ascent method embedded in it. Other improvements to the Held and Karp method were suggested by Hansen, Krarup [Ref. 10] and Houck, Picard, Vemuganti [Ref. 11]. Bazarra and Goode [Ref. 12] did further work on Held and Karp's approach using optimization of a lagrangian dual in lieu of solving integer linear program. They proposed a branch and bound scheme with subgradient optimization of the dual to transform the one-tree into the optimal tour. The one-tree method as described is exact and may require extensive computation time.

D. TOUR IMPROVEMENT

The Tour Improvement method assumes that an arbitrary tour is available. The method operates by perturbing or exchanging arcs in the tour until a better tour is found. The Tour Improvement method terminates when a better tour cannot be found. In its simplest form, it is possible for this method to check $(n-1)!/2$ arc exchanges for the symmetric TSP before terminating (i.e., it investigates all available answers).

Dantzig, Fulkerson, and Johnson [Ref. 13,14] proposed a algorithm which starts with an arbitrary tour and uses integer linear programming to improve the tour. The TSP is transformed into an integer linear program and solved using the simplex method. Dantzig, Fulkerson, and Johnson [Ref. 15] mention that the number of constraints needed to characterize the problem is 'astronomical.' Instead of using all the constraints, they begin by using a relaxed version

of the problem and add constraints as needed to the integer linear program to maintain feasibility. In their method infeasibility appears as 'loop' or subtour. To continue solving the problem a constraint is added which removes the subtour yet does not eliminate any of the available tours. This algorithm requires the addition of constraints during the solution process to maintain feasibility. Several researchers have suggested improvements on Dantzig, Fulkerson, and Johnson's method for solution of the TSP. [Ref. 16,17,18,19]

Croes [Ref. 20] proposed a heuristic whereby a subset of all possible transformations is tested, a transformation being the transformation of one tour into another tour. He called these transformations 'inversions' because they inverted the sequence of nodes in part of the tour to create a possible improvement. Lin [Ref. 21] later describes Croes 'inversion free tours' as 2-optimal tours and goes on to describe k-optimality where k is some fixed number less than n. The general idea is to exchange k arcs iteratively in the tour while testing for improvements. Lin and Kernighan [Ref. 22] presents a modified k-opt algorithm where k is not fixed. In this algorithm k may vary from 2 to n. The floating k-opt algorithm is also described by Christofides and Eilon. [Ref. 23]

The advantage of the 'inversions' or k-opt solution method is that for a given problem, the memory needed to solve the problem is fixed. There are no constraints which must be added to the problem, and the decision rule used is simple. The k-opt method is exact, but may be terminated before optimality is reached to provide a satisfactory (suboptimal) solution. This generally reduces run time and still provides a good solution to the problem. The k-opt method may also be programmed to require very little user interaction; in the best case the user need only enter the

distance matrix for the nodes desired in the tour. The k-opt algorithm may require extensive computation time, particularly if during solution k is found to be greater than 2 on many of the iterations.

This chapter has presented a general survey of some the solution methods for the TSP and some algorithms associated with these solution methods. The next chapter will discuss which of these solution methods were employed to solve the bouytender problem. The buoytender routing algorithm will then be presented and discussed in detail.

IV. BUOYTENDER ALGORITHM

This chapter will discuss the algorithm with which we propose to solve the buoytender problem. The reasons for the algorithm's selection and some background on its development will be presented, together with a detailed description of how the algorithm addresses the buoytender problem.

A. ALGORITHM SELECTION

The algorithm is a combination of two of the previously mentioned solution methods in that it combines a Tour Building heuristic with a Tour to Tour Improvement heuristic. The concept of combining a Tour Building heuristic with a 2-opt or 3-opt heuristic was proposed by Golden et al. [Ref. 24] as a relatively fast computational solution method which would provide an optimal or near optimal tour. Their 'composite' algorithm is the foundation on which the buoytender algorithm was developed.

The heuristics used in the buoytender algorithm are the nearest neighbor algorithm and the k-opt algorithm. The nearest neighbor algorithm is used to construct an initial tour while a version of the k-opt proposed by Lin and Kernighan [Ref. 22] is used to improve this initial tour. The nearest neighbor and k-opt heuristics were selected for use in the buoytender algorithm because very little interactive work is required, the interactive work is simple, no constraints need to be added to maintain feasibility, computational time is reasonable, and the algorithm operates with a fixed amount of memory.

The buoytender algorithm as presented requires very little user interaction. The nearest neighbor algorithm

needs only the entry of the distance matrix for the problem and the selection of a starting node. The k-opt algorithm needs the same distance matrix and an initial tour. The heuristics' decision processes will provide a feasible and possibly optimal solution in a reasonable period of time. The nearest neighbor algorithm was selected to construct the initial tour because it is computationally faster than insertion techniques, and provides a good initial solution to the problem. [Ref. 25]

A critical factor in solving the TSP on a microcomputer is the memory required by the algorithm used. The amount of memory required for solution of a TSP of size n is fixed for the nearest neighbor and k-opt heuristics. With known memory requirements for the heuristics, a program can be developed to fit the Coast Guard Standard Terminal. The nearest neighbor algorithm requires an n by n distance matrix and an n array in which to store the tour as it is constructed. The k-opt algorithm requires the same storage as the nearest neighbor plus a n by n decision matrix, and a 2n array for recording nodes selected for inversion in the tour.

B. PROGRAMMING THE ALGORITHM

The buoytender algorithm was developed as a series of modules which were then programmed as subroutines in the operating program. The algorithm was programmed in FORTRAN for operation on the Coast Guard Standard Terminal. FORTRAN was selected because it is efficient and a majority of programmers are familiar with it.

There are some differences between the heuristics in the literature and the buoytender algorithm. The nearest neighbor algorithm as used by the buoytender algorithm is the same as outlined by Rosenkrantz, Stearns, and Lewis

[Ref. 2]. The k-opt heuristic used in the buoytender algorithm is a modified version of the heuristic presented by Lin and Kernighan [Ref. 22]. There are three major differences between the k-opt in the bouytender algorithm and the k-opt suggested by Lin and Kernighan. First, the Lin and Kernighan k-opt heuristic selects all possible nodes which will result in a shorter tour before creating the improved tour. The buoytender algorithm improves the tour by exchanging nodes whenever a favorable selection is found. This improvement-as-you-go procedure results in a simplifed selection process for the next set of nodes to be tested. Second, the Lin and Kernighan k-opt heuristic has an added facility for 'limited backtracking' used when a particular exchange gives an improvement of zero. The backtracking procedure searches until a gain greater than zero can be found which improves the tour. In the interest of shortening computational time, the buoytender algorithm does not 'backtrack' when a gain of zero is found, but instead treats zero gain as no improvement and continues with the next selection. The third major difference between the Lin and Kernighan k-opt heuristic and the buoytender algorithm is related to what Lin and Kernighan call 'reduction.' Their heuristic, after producing several locally optimal tours, checks for arcs which appear in each of the tours. These 'good' arcs are not allowed to be broken in further computations for other locally optimal tours thereby reducing the number of links to be checked for improving the tour. While decreasing run time, this procedure requires more memory and is therefore omitted from the buoytender algorithm.

The buoytender algorithm can be divided into three basic operations. First, the algorithm selects a random sample of k nodes, $k<n$. This random sample of nodes becomes the set of starting nodes for the next major operation in the algorithm, to create k nearest neighbor tours. The third

operation is to use the k nearest neighbor tours as initial tours for the k-opt improvement. The best of the k-opt improvement tours is presented as the solution to the problem.

A further modification was made to the algorithm based on some computational results obtained from use of an early version of the algorithm. During programming of the buoy-tender algorithm several benchmark statistics were obtained from the program. Two of these statistics were the initial tour distances produced in the nearest neighbor phase, and the distances of the final tour solutions produced by the k-opt phase of the algorithm. There seemed to be a relationship between the tour produced by the nearest neighbor phase and the improved tour produce by the k-opt phase. A least-squares linear fit of the initail tour distance vs. the final tour distance was done for four data sets. In each case the fit prcduced a positive slope. This implies for a relatively large nearest neighbor tour, the final tour from the k-opt phase will be relatively large. A relatively small nearest neighbcr tour will produce a relatively small tour frcm the k-opt phase. Using this information the buoy-tender algorithm was further modified to reduce computation time without compromising too much of its ability tc produce optimal solutions. For a set of k nodes, k<n, the nearest neighbor algorithm provides k initial tours. The set of k nodes is selected at random from the population of n nodes. Then for a set of L ncdes, L<k, the k-opt algorithm is run to obtain L final tours. The set of L nodes are drawn from the k nodes which produced the L shortest initial tours. The shortest final tour from the L final tours produced by the k-opt phase will be the algorithm's solution to the problem. The final version of the algorithm has a greatly reduced operating time over the initial version.

C. ALGORITHM DESCRIPTION

In this section, a detailed description of the buoytender algorithm is given. In this description references will be given to the program subroutines so that the reader may associate the portion of the algorithm under discussion with its operational counterpart in the program.

The buoytender algorithm begins by reading a file from mass storage containing the number of aids (nodes) in the problem, the distance matrix, and the names of the aids (subroutine OBTAIN). A program MATUTL.FOR, which assists the user in creation of this data, can be found after the listing of the buoytender route selection program, ROUTE.FOR.

After obtaining the data the algorithm selects a random sample of nodes, based on the size of n, from which it will create tours (subroutine NODE). The function which selects the size of the random sample is

$$\text{Sample Size} = 5 + \lceil 2(\text{Log}(n-4)) \rceil .$$

The function is designed to capture almost the entire population for testing when the problem is small and to capture only a small portion of the population being tested when the problem is large. This produces a sample size which increases the probability of obtaining the global optimum when the problem is small yet produces a sample size which will maintain reasonable computation times when the problem is large.

After determining the sample size, a random sample of nodes is taken. Since there is no random number generator for FORTRAN installed on the Coast Guard Standard Terminal, a pseudo-random number generator was added to program (function RANDOM). The coding for the generator comes from Wichmann and Hill [Ref. 26] ; it is described by its

authors as an efficient psuedo-random number generator having a cycle length of 2.78×10^{13} . Nodes are sampled at random without replacement from the integer population of 1 to n for use as starting nodes for the nearest neighbor phase.

The algorithm then moves into the nearest neighbor phase (subroutine NBR). A tour is created for each node in the random sample, with the node from the sample as the starting node of the tour. The algorithm selects the nearest node to the starting node for the next entry in the tour construction. The algorithm then iterates selecting a node not yet in the tour which is nearest to the previously selected node as the next entry in the tour. This process continues until all nodes have been selected, at which time the last node is connected to the starting node to complete the tour, and the tour distance is computed. After all the initial tours have been constructed they are ordered by their total distances from shortest to longest. The five shortest tours are retained for possible improvement in the k-opt phase.

A decision matrix is created which designates which distance arcs in the distance matrix are presently members of an initial tour (subroutine MARKD). The algorithm then proceeds to step through each node of the tour testing it for k-optimality.

The tour is 'prepared' at each node to be checked for k-optimality (subroutine TRPREP). This process simplifies some of the later operations. Here, it is helpful to consider the tour as a circle with n postions on its circumference (see Figure 4.1), and to let positions on the circular tour be denoted by p(i) where i=1,...,n. A node in the tour is represented as a position on the circle. TRPREP places the starting node in p(1) and places the remaining nodes in the tour at p(2) through p(n) where (i) is the node's position in the tour.

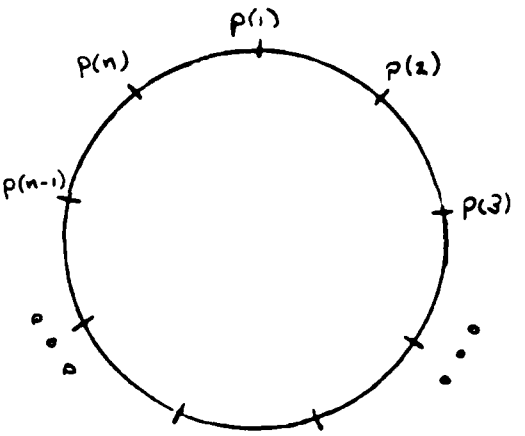


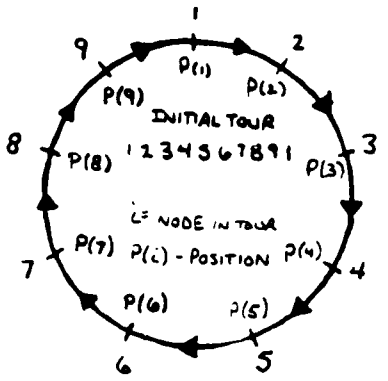
Figure 4.1 Circular Representation of a Tour

Since floating k-opt algorithm is in essence a series of 2-opt iterations a description of the simpler 2-opt principle will be presented first. Figure 4.2 shows the 2-opt principle in graphic detail. In Figure 4.2 (A), the tour is represented in circular form. The tour is now perturbed to try and find an improvement. To perturb the tour in the 2-opt procedure, two arcs will be broken in the existing tour and two new arcs will be selected as incoming to form a new tour. This provides a gain function which can be used to check for improvement. The gain function is

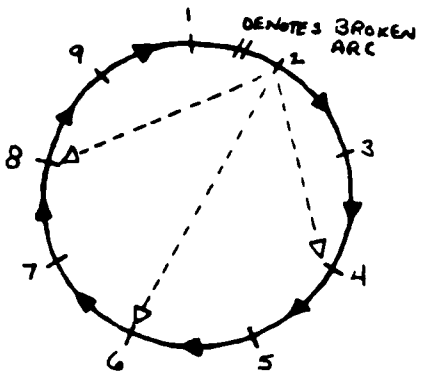
$$G = \sum_{i=1}^{2}(\text{broken arcs}) - \sum_{i=1}^{2}(\text{incoming arcs}).$$

If G is positive this set of arc exchanges will improve the tour.

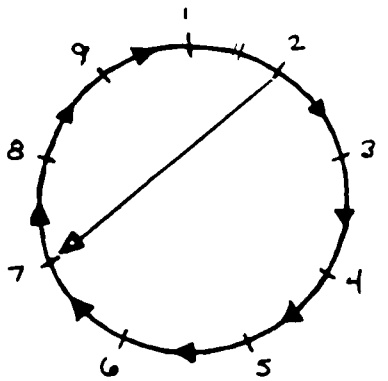
To perturb the tour, one of the arcs incident to node being checked for 2-optimality must be broken or removed from the tour. In the buoytender algorithm the node being checked for k-optimality will always be in p(1) of the circular tour and the arc being broken will always be between p(1) and p(2) of the circular tour. An incoming arc must then be selected to replace the broken arc. This incoming arc must be selected from those arcs with a starting node which is the same node as the end node of the



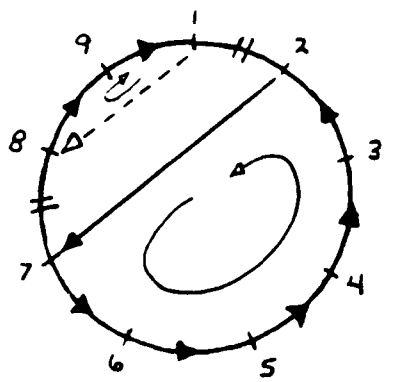
(A) Circular Tour



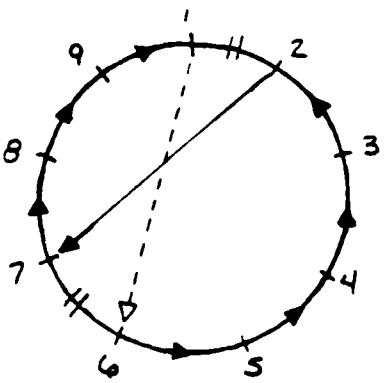
(B) Search for Incoming Arc



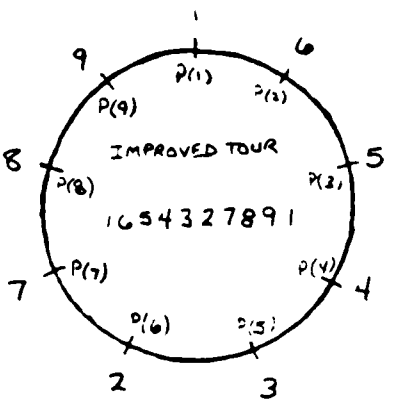
(C) Incoming Arc Selected



(D) Wrong Incident Arc Broken



(E) Feasible Arc Exchange



(F) Circular Tour

Figure 4.2 Graphic Description of 2-opt

previously broken arc. Figure 4.2 (B) shows some of the available selections for the incoming arc. The incoming arc

is in general the shortest available arc, but may be any of the available arcs which maximize G. This subtour must be broken and another incoming arc must be selected so as to reconstitute a tour. Therefore, one of the tour's arcs incident to the end node of the first incoming arc must be broken. The end node in Figure 4.2 (C) is at p(7). There is only one of two possible tour arcs which may be broken to allow the incoming arc to become part of the tour. Figure 4.2 (D) shows the selection of the wrong incident tour arc for removal. The breaking of this arc and the subsequent reconnection to the node being checked for 2-optimality does not result in a tour. Instead, two subtours are created giving an infeasible situation. Figure 4.2 (E) shows the breaking of the proper incident arc and subsequent reconnection to the original node to create the 'improved tour'. Also notice that some of the arc directions in the original tour must be reversed to provide a consistent direction of movement through the tour. Reversing these arc directions is the same as inverting a sequence of nodes in the tour. The result is shown in Figure 4.2 (F) as a new 'improved' circular tour.

The 2-opt procedure is carried out using each node in the tour as the starting node checked for 2-optimality. One interation of the 2-opt procedure through all n nodes will produce a tour which is 2-optimal.

The k-opt algorithm is very similar to the 2-opt procedure described above. Instead of just reconnecting the tour as the 2-opt procedure does (p(6) to p(1)), the k-opt checks to see if another incoming arc, starting at the end node of the last broken arc, can be selected to further improve the tour (Figure 4.3). The procedure interates until no further improvement can be found. A k-opt, using an example with k equal to four, might look like Figure 4.4 The floating k-opt algorithm has the ability to vary k during the k-opt

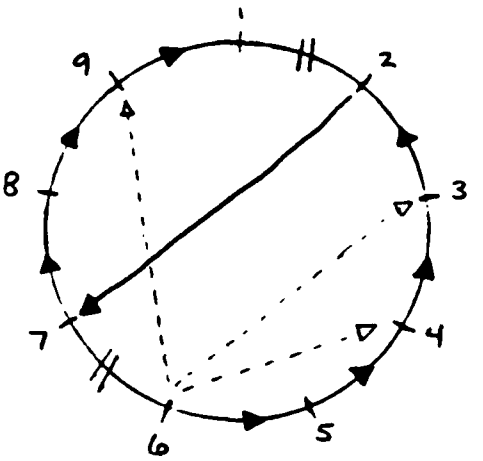


Figure 4.3 K-opt Selecting Next Incoming Arc

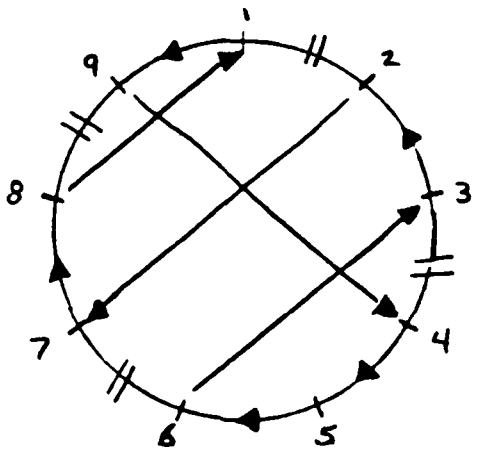


Figure 4.4 Example of 4-opt

solution process. This is more desirable than fixing k in the solution process. For example, if k were fixed at four, the algorithm can only check four arc exchanges for improvement but the problem may need a five arc exchange to produce the optimal solution. By allowing the k-opt to vary k, the probability of the algorithm finding the optimal solution is increased. The floating k-opt is bounded between 2-optimality and n-optimality. The k-opt cannot be greater than n-optimal because an n-opt exchange creates the optimal tour.

The k-opt procedure in the buoytender algorithm operates by checking each node in a tour for k-optimality. The algorithm, using subroutine TRPREP, creates the circular tour for the node being checked for k-optimality. The circular tour is prepared so that the initial arc being broken will always be between p(1) and p(2). The algorithm checks the circular tour at the starting node for k-optimality. If no improvement can be made breaking the first arc between p(1) and p(2) then the circular tour's direction is reversed (subroutine REVTOR) so that the alternate incident arc is placed between p(1) and p(2) of the circular tour and this tour is checked for k-optimality.

The breaking of the arc between p(1) and p(2) requires an incoming arc to replace it (subroutine SELCTY). The algorithm selects the five shortest available arcs which originate at position two of the circular tour and terminate at another position on the circular tour. The arc from p(2) to p(3) is not available since it is already in the tour, and the arc from p(2) to p(1) is not available since it is the arc just broken to improve the tour. For each of the selected incoming arcs another arc must be broken to eliminate a subtour (as in the 2-opt procedure). It is then necessary to reconnect the arcs to reconstitute the tour. This results, for each of the selected arcs, a set of two incoming arcs and two outgoing arcs. Using the previously defined G, each of the five sets of arcs is checked to see which maximizes G. If G>0 then the exchange or inversion is made (subroutine ADJTOR), otherwise no exchange is made.

If an exchange is made the algorithm goes back and searches for a new incoming arc from the end point of the last broken arc. Subroutine ADJTOR has inverted the tour so that the last arc used to reconnect the tour is now between p(1) and p(2) of the circular tour. Subroutine SELCTY is again used to select the next best available incoming arc as

described above. The algorithm interates through selecting an incoming arc and inverting the tour as long as improvement, i.e. positive gain, can be found.

When improvement can no longer be found the direction of the tour is reversed at the last node where improvement was found, and the tour is tested again for possible further improvement. When nc further improvement can be found and the reversed tour has also been tested, the algorithm then increments to the next node in the tour and starts the k-opt procedure again testing this node for k-optimality. Each node in the tour is used as a starting node in the k-opt procedure. When all nodes in the tour have been tested for k-optimality the k-opt phase ends and the resulting tour is stored.

After each of the five shortest initial tours have been tested for k-optimality, the shortest of the improved k-opt tours is selected as the solution to the buoytender problem. Figure 4.5 is a flowchart of the buoytender algorithm.

The next chapter discusses the results of some test problems solved by the algorithm and makes concluding remarks about the algorithm.

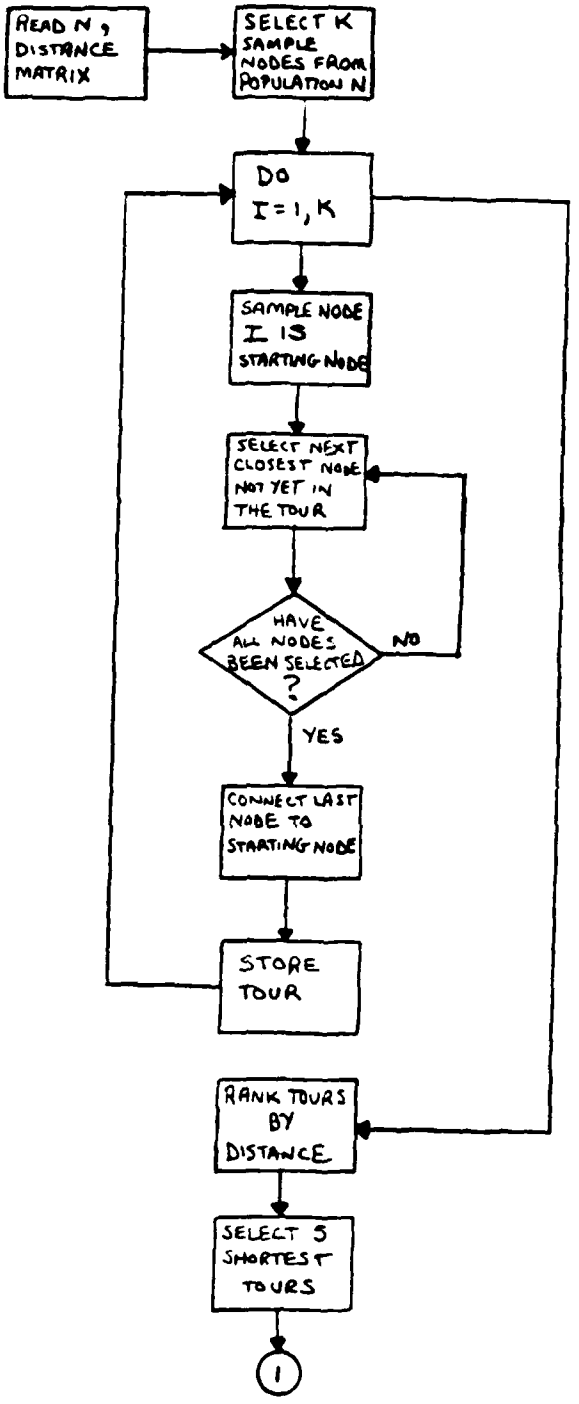


Figure 4.5 Algorithm Flowchart Section 1

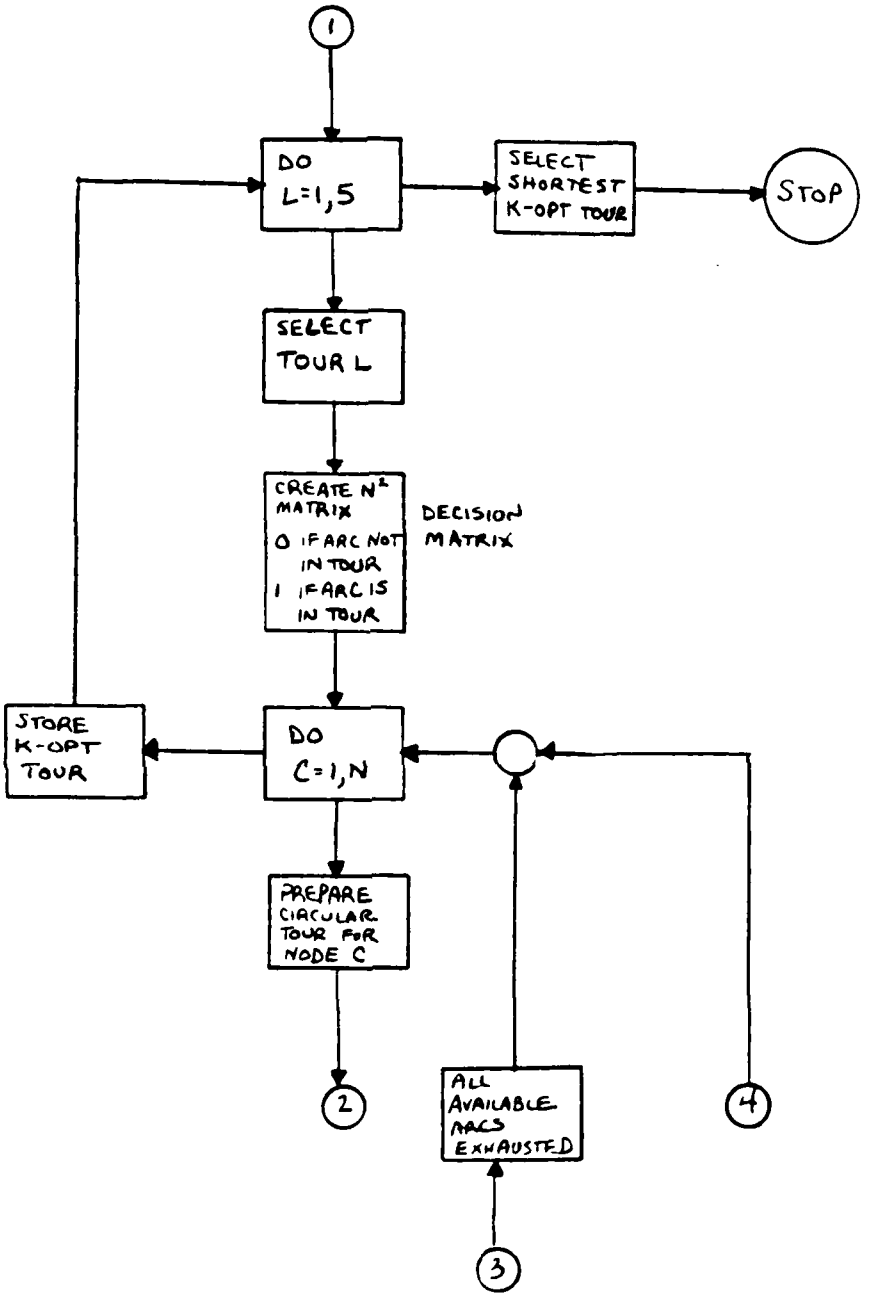


Figure 4.6 Algorithm Flowchart Section 2

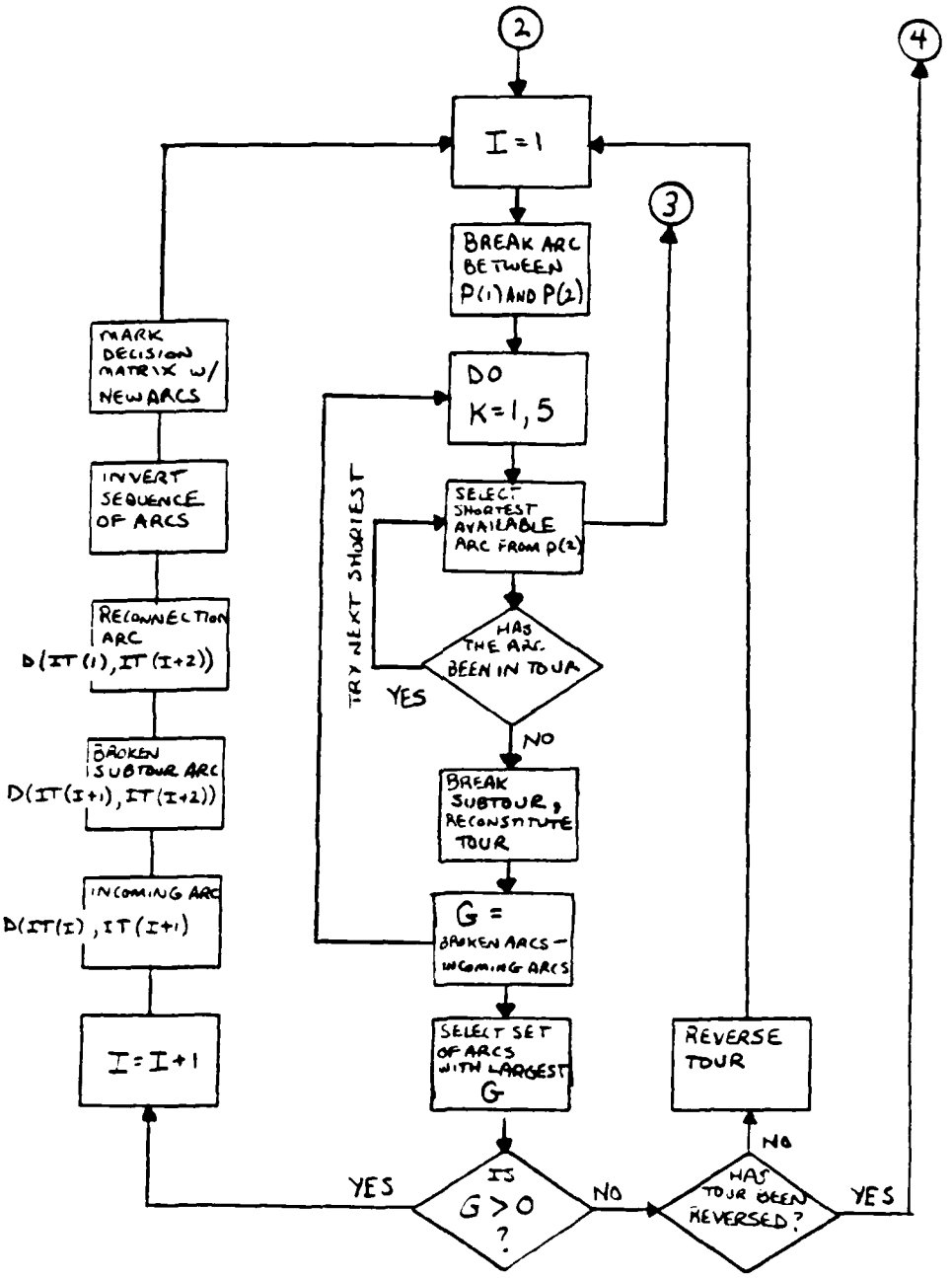


Figure 4.7 Algorithm Flowchart Section 3

V. PROGRAM RESULTS AND CONCLUSIONS

This chapter will discuss some results obtained from using the buoytender route selection program, including results from an actual problem from the U.S. Coast Guard Cutter BLACKHAW. All program results were obtained using the Coast Guard C3 Standard Terminal. The chapter will conclude with some observations regarding the bucytender problem, the algorithm, and the test results.

A. PROGRAM RESULTS

To program and test the buoytender algorithm it was necessary to obtain problems with known optimal solutions. Initially, a very simple and small problem with nine nodes was created. This test problem's optimum tour is the most obvious tour obtained by inspection of the problem since the nodes are arranged in an almost circular pattern. Connecting these nodes following this nearly circular path results in the optimum solution. Several alternate tours were checked by hand computation and were found to be suboptimal. Therefore, the near circular path for this test problem is assumed to be optimal. When the program was tested with this test problem, the program produced this hypothesised optimum solution with a computation time of 13 seconds.

With the knowledge that the program appeared to be operating correctly on the initial test problem, further testing was necessary to check the program's ability to solve other problems. Since the buoytender route selection problem is a symmetric case of the TSP, the literature was searched for test problems. Three articles [Ref. 5,20,27] yielded eight

symmetric TSP's of various sizes. These eight literature problems ranged in size from 5 to 57 nodes with known optimal answers. The program, when tested with these problems, produced solutions which ranged from the optimal to within 3.5 percent of the optimal distance.

Another problem was created by the random selection of 70 points on a Euclidean plane. A distance matrix was computed for these 70 nodes and this problem was solved by the program. The problem's optimal solution is not known, but the problem was run to ensure the program could handle the maximum problem for which it was designed, and to produce a sample solution time for this maximum sized problem.

With evidence that the program was solving test problems, a further test was conducted using an actual buoytender routing problem, with data provided by the U.S. Coast Guard Cutter BLACKHAW (WLB-390), homeported in San Fransisco. The aids to navigation run data was for a trip scheduled from 9 April 1984 to 16 April 1984. With the experience and historical data this vessel has available from servicing these aids for many years, it is hypothesized that the route which was scheduled for this ATON run would be optimal or nearly optimal. The planned route for the ATON run had a distance of 455 nautical miles. When this problem was entered and solved by the buoytender program, the solution was also 455 nautical miles. Although the distances were the same, the scheduled route and the program route differed slightly in their sequence for visiting the aids. This difference may be attributed to the fact that, as stated in Chapter II, the triangle inequality may be required to be a strict equality on some of the arcs. This difference is possible since the buoytender may, due to the strict equality on some arcs, be required to 'pass' a previously serviced aid to complete the route. Figure 5.1 shows

a five node example of this situation. This suggests there may be several sequences of nodes which will provide an optimal solution to the buoytender problem.

TABLE 1

Tours for BLACKHAW Problem

Route as Scheduled by the Buoytender		Route Produced by the Program	
1	YB ISLAND	1	YB ISLAND
2	MONTARA LWB10A	20	MOSS LBB MLA
3	AN ISLAND LWB8	19	MONTRY BY LB B
4	SANTA CRUZ MB	18	MONTRY HBR MB
5	P BLANCAS LWB4A	17	PT PINOS LWB2
6	SAN SIMEON LBB1	16	PT CYPRESS LGB6
7	VON HELM R LGB4	5	P BLANCAS LWB4A
8	ESTERO BY GB10E	6	SAN SIMEON LBB1
9	SOUZA R LGB	7	VON HELM R LGB4
10	WESTDAHL R LBB1	15	ESTERO BY LWB
11	PT SANLUIS LWB3	8	ESTERO GB10E
12	LANSING R B	14	MORRO BY LBB1
13	PT BUCHON LWB2	13	PT BUCHON LWB2
14	MORRO BY LBB1	10	WESTDAHL R LBB1
15	ESTERO BY LWB	9	SOUZA R LGB
16	PT CYPRESS LGB6	12	LANSING R B
17	PT PINOS LWB2	11	PT SANLUIS R B
18	MONTRY HBR MB	4	SANTA CRUZ MB
19	MONTRY BY LB B	3	AN ISLAND LWB8
20	MOSS LBB MLA	2	MONTARA LWB10A
1	YB ISLAND	1	YB ISLAND

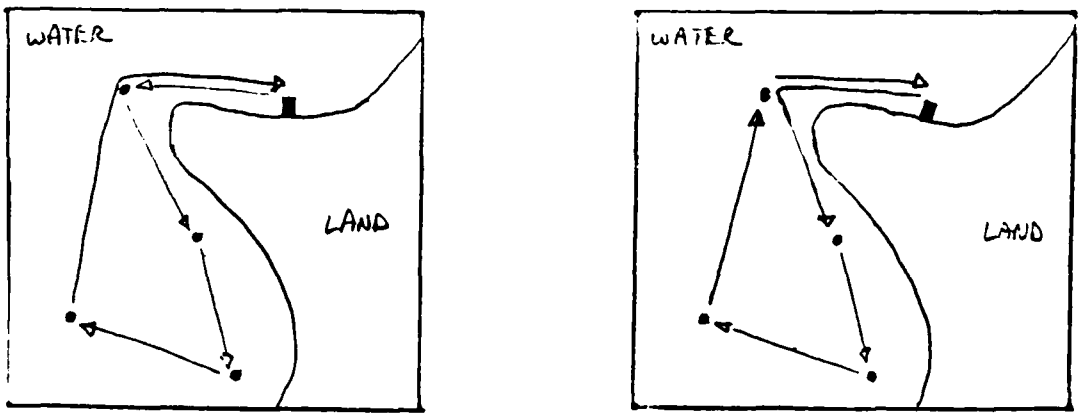


Figure 5.1 Multiple Optimal Solutions

TABLE 2

Results of Test Problems

Problem & (Ref.)	Size	Sclution Time(Sec)	Tour Dist. (Miles) Program	Optimal	Solution Quality*
Karg(27)	5	5	148	148	100%
Test(auth.)	9	13	22.95	22.95	100%
Barachet(27)	10	17	378	378	100%
BLACKHAW	20	50	455	455	**
Croes(20)	20	52	254	246	103.25%
HeldKarp(5)	25	72	1711	1711	100%
Karg(27)	33	121	10929	10861	100.62%
Dantzig(27)	42	188	705	699	100.86%
HeldKarp(5)	48	238	11847	11470	103.28%
Karg(27)	57	340	13367	12956	103.17%
Max(auth.)	70	494	6791.07	unk.	.unk

* 100% is optimal; 103.28% is 3.28% greater than optimal.
** The distance is hypothesized to be optimal.

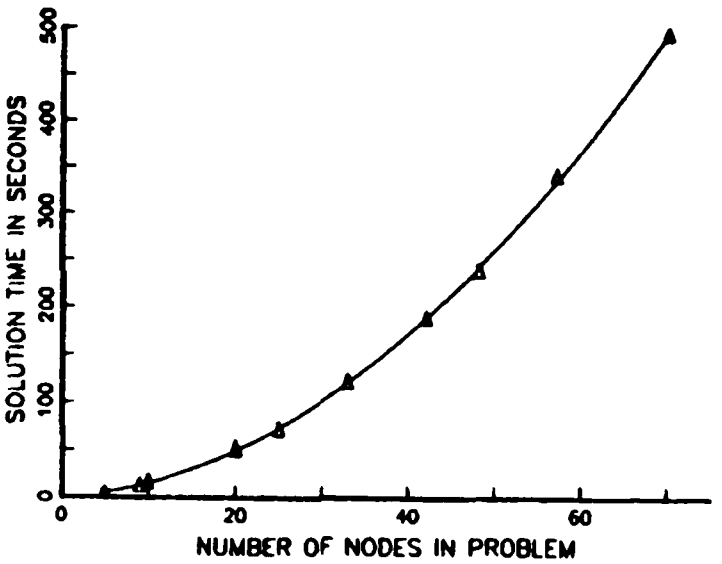


Figure 5.2 Least Squares Fit of Size vs. Solution Time

Table 2 presents a summary of the eleven test problems showing problem sizes, solution times, and the quality of the solutions produced by the program. All of the run times are reasonable by the criteria stated in Chapter II. The largest problem, 70 nodes, had a solution time of 494 seconds, or roughly a little more than eight minutes.

Ccmputation times for the eleven problems were plotted against problem size, as shown in Figure 5.2 A least-squares polyncmial fit of the data points is

$$\text{Time} = .584 + .62n + .0914n^2 + .00000994n^3$$

and this curve is alsc shown in the figure.

B. CCNCIUSIONS

The buoytender route selection algorithm provides another useful way of scheduling ATON routes. As a tool for the operator to assist in planning the route to be taken, the program should produce optimal or near optimal solutions for prcblems up to a size of 70 aids. The user may accept this route as is or may desire to modify the route based on circumstances requiring human judgement. Appendix A is a user's guide for the matrix utility and buoytender route selection programs.

The buoytender route selection algorithm, with the nearest neighbor and k-opt heuristics, provides a quick and satisfactory soluticn to a symmetric traveling salesman problem cn problems up to a size of 70 nodes. The prcblem may have several routes with different sequences for visiting the aids, but the routes may all have the same distance.

It is hoped that this algorithm will be of use to the fleet cf U.S. Coast Guard buoytenders. As a decision aid it may help the operators of buoytenders obtain optimal or nearly optimal routes to service their aids, possibly saving time and fuel.

APPENDIX A
USER'S GUIDE TO PROGRAMS

The intention of this appendix is to provide instructions for the use of the buoytender route selection program and utility programs. It should be noted that prior to using the buoytender route selection program, the user will need to use the matrix utilities program listed at the end of this appendix to prepare a data file.

The buoytender route selection program and the matrix utilities program are written in Convergent FORTRAN. Convergent FORTRAN is FORTRAN 77 compatible. The programs should be typed in as presented and linked to the operating system. The Convergent FORTRAN manual has instructions concerning compiling and linking. Recommended names for the run files are ROUTE.RUN and MATUTL.RUN. Both programs will operate on either a IWS or AWS station with 264K memory or more. The programs run about three time slower on an AWS terminal than on an IWS terminal.

The user will be required to enter text and numerics into the programs. All numeric entries are to be integer entries (no decimal point) with the exception of the distance entries. All distance entries require a decimal point entry for proper input.

MATUTL.RUN is capable of entering problems up to a size of 100 aids. ROUTE.RUN is designed to handle problems up to a size of 70 aids. MATUTL.RUN allows the entry of a large matrix of aids and includes a utility which allows the user to select a subset of this large matrix for use in the program ROUTE.RUN. This allows the user to type in one large matrix and then create smaller matrices from the larger as needed. This feature will preclude the user from

typing in a distance matrix every time he desires to run a problem.

A. USE OF MATUTL.RUN

The matrix utilities program is designed for creating and editing data files which will be used in the buoytender route selection program. A data file will contain the size of the problem n, the distance matrix for the problem, and the names of the aids to be visited.

The size of the problem, n, will be the number of aids to be visited plus one. The plus one is for the port from which the buoytender will start and end the ATON run.

To create the distance matrix some extensive plotting is required. Most buoytenders have a set of tracklines they regularly follow to visit their aids. In most cases these existing tracklines will be sufficient to use in the distance matrix. To create a distance matrix for entry into a data file it is recommended the user use the format shown in Figure A.1 . Distances should be computed in nautical miles and may range from .01 nm to 9999.99 nm. Figure A.1 is an example of a problem of size nine.

Since the problem is symmetric, the user need only calculate and enter the distances as shown in Figure A.1 . The matrix utilities will automatically complete the rest of the matrix entries. The distance you should compute should be the shortest possible distance from the row entry to the column entry. Using the example in Figure A.1 the entry in row 3, column 6 is the shortest possible trackline from aid A to aid E. Once the entries for the distance matrix have been prepared the user is ready to use MATUTL.RUN.

MATUTL.RUN has some simple menus and some descriptive comments to help the user. The menus require a numeric response for the selection of an option. The first menu presents the utilities available to the user, Figure A.2 .

	1	2	3	4	5	6	7	8	9
1 HOMEPORT		.9	1.	1.9	5.1	8.85	9.2	8.4	7.
2 AID A			.6	1.	4.7	8.45	8.8	7.9	6.45
3 AID B				1.	4.15	7.9	8.25	7.4	6.
4 AID C					4.2	7.95	8.3	7.2	5.6
5 AID D						3.75	4.1	3.4	2.7
6 AID E							.35	3.8	5.1
7 AID F								3.9	5.4
8 AID G									2.3
9 AID H									

Figure A.1 Sample Distance Matrix

```
1 - create a distance matrix
2 - extract a small matrix from a larger matrix
3 - correct a previously stored matrix
4 - create a hardccpy of a distance matrix
5 - list names with their row number in the matrix
6 - exit this prcgram

What is your selection?
```

Figure A.2 MATUTL.RUN Main Menu

1. Create a Distance Matrix

This utility is for the initial entry of a distance matrix into a data file. The user will be asked for the size of the problem, the homeport and aids names, and the

distances. The size of the problem is the number of aids plus one, for homeport. The utility will next ask for names of homeport and the aids to be visited. These names will be associated with a row and column number of the distance matrix. The names are text entries up to a maximum of 15 characters.

The utility will next ask for distances. It will ask for the distance from a row number to a column number. If the user sets up the distance matrix as shown in Figure A.1 he will be able to then just read each row across its columns while entering the data. These distances must be real numbers. The distances should be nautical miles and may range from .01 nm to 9999.99 nm. Since these entries are real numbers there must be a decimal point in each entry. If, for example, the user needs to enter two nautical miles, he should enter it as 2. <return>. Be careful with these entries; if a decimal point is not entered the distance will be off by a factor of 100. For quickly and easily entering data the numeric keypad on the keyboard may be used and <next> may be used in place of <return> to enter the data. If an error is made while typing in numbers and <return> or <next> have not yet been pressed, then the error may be corrected by using the back-space key and retyping the number. If an error is made and <return> or <next> have already been pressed, then the user should note the row and column number of the error and use the correction utility to correct the entry.

When the user finishes typing all entries the program will prompt for a filename in which the data will be stored. This name may be up to 15 characters long. If for some reason the user needs to end the terminal session before the entire matrix is entered, data entry may be stopped by typing in a negative distance. The user will then be asked for a filename for the partial data file. To

complete the partial data file the user must use the correction utility.

2. Extract a Matrix

This utility is designed so that the user may choose a subset of aids from a larger matrix. The utility will create the proper data file for a subset of aids.

The user should have an idea of all possible aids which may be visited on a general buoytender run (i.e. the spring run, the north run, the inside run, etc.). The user should create a distance matrix using the create utility of all aids which could possibly be visited on a general run. Usually only some portion of these aids will be visited on an actual run (for example on a particular spring run only 90% of the aids which might be visited on this run need to be visited). In this case the user creates a matrix of all aids which might be visited on a run and then each time this run is scheduled the user can then use the extract utility to quickly and easily create a data file for those aids which need to be visited.

Before using the extract utility it is suggested the user first use the list utility on the larger or initial data file. This will provide the user a list of aid names in the data file and present them with their corresponding row number.

The extract utility will prompt the user for the name of the data file containing the general run's aids. The user is then asked for the size of the new problem whereby the user enters some m, m less than n in the initial data file. The utility then asks the user m times for row numbers from the initial matrix. After m row numbers have been entered the utility creates the new data file and askes the user for a filename for this new data file.

3. Correct a Previously Stored Matrix

This utility is to correct entries in a distance matrix or tc complete a partially entered data file. The utility has a menu which askes if the user wishes tc make single entry or sequential entry corrections. The single entry mode asks the user for the row number and column number of the entry desired to be corrected. The user then is asked for the new distance and is returned to the correction menu where he may select to make another correctien or exit the utility.

The sequential corrections are similar to the entry of distances in the create utility. The user is asked for the starting row number and starting column number where correcticns are to begin. The utility asks for the remaining column entries on the starting row and then will proceed to the next row until entries are terminated. The utility automatically terminates at the end of the matrix or the user may terminate the sequential mode early by entering a negative number. If a negative number is entered it will not appear in the distance matix. Terminating the sequential mode returns the user to the corrections menu for another correction or to exit the correction utility.

Upon exiting the correction utility the user is asked for a filename for the corrected file. This filename may be up to 15 characters long.

4. Prepare a Distance Matrix for Hardcopy

This utility is for preparing a hardcopy of a distance matrix from a stored data file. The data files are saved as sequential files, (i.e., the data files are a string of numbers). This utility will construct a file which will resemble a matrix. Due to the width and length limitaticns of the printer and paper, the matrix is printed

in sections which will have to be cut and pasted together to display the whole matrix. A hard copy of the whole matrix will provide the user with a means of checking to see if data entries are correct. If an entry error is found, the matrix also helps the user to locate the row and column number of the error.

The utility asks the user for the name of the data file from which a distance matrix is needed. The utility creates the hardcopy file and then prompts the user to obtain a hardcopy by using the FORMAT command to print out file MATPRT.DATA.

5. List Aid Names

This utility provides a hardcopy of the aid names from a data file. This may be useful to the user when using the extract utility. The user is asked for the name of the data file from which the list shall be made. The utility then prepares a file containing the row numbers and their associated aid names. The user is prompted to use the FORMAT command to obtain a printout of LIST.DATA.

These matrix utilities are for assisting the user in creating and manipulating data files and distance matrices. A correct data file is necessary for the proper operation of the buoytender route selection program, ROUTE.RUN.

B. USE OF ROUTE.RUN

The user of ROUTE.RUN is very simple once a data file has been created. The user runs the ROUTE.RUN program and he is prompted for the name of a data file. The user is then asked for the name of an output file for the results. The program obtains the size of the problem from the data file and will compute an approximate solution time. This approximate solution time is based on results from some test

problems solved by the program and it assumes the user has version 8.01 of the operating system. If another version of the operating system is used the solution time may differ.

After displaying the approximate solution time the program reads the rest of the data file and proceeds to do the necessary computations. The user, depending on the size of the problem, will have just enough time to get a cup of coffee before obtaining the solution. The solution will be displayed on the screen and will also be placed in the file previously named for results, from which a hardcopy may be made. The user now has a possible route to take for that particular aids to navigation run.

APPENDIX B

SAMPLE OUTPUT OF BUOYTENDER ROUTE SELECTION PROGRAM

The solution for data file BLACKHAW.MAT
Total tour distance = 455.00
The tour is as follows

1	YB ISLAND
20	MOSS LBB MLA
19	MONTRY BY LB B
18	MONTRY HBR MB
17	PT PINOS LWB2
16	PT CYPRESS LGB6
5	P BLANCAS LWB4A
6	SAN SIMEON LBB1
7	VON HELM R LGB4
15	ESTERO BY LWB
8	ESTERO GB10E
14	MORRO BY LBB1
13	PT BUCHON LWB2
10	WESTDAHL R LBB1
9	SOUZA R LGB
12	LANSING R B
11	PT SANLUIS R B
4	SANTA CRUZ MB
3	AN ISLAND LWB8
2	MONTARA LWB10A
1	YB ISLAND

APPENDIX C

LISTING OF BUOYTENDER ROUTE SELECTION PROGRAM

```
C     MAIN PROGRAM *** BUOYTENDER ROUTE SELECTION ***
      IMPLICIT CHARACTER*15 (A)
      REAL G,TDIS,TORDIS,DIS(15)
      INTEGER I,J,L,C,R,NUM,NOD,SAM(15)
      CHARACTER*15 FNAME,ONAME
      COMMON /NAME/ ANAME(70)
      COMMON /WALK/ N,ITOUR(71)
      COMMON /TDST/ TD(70,70)
      COMMON /INVT/ IT(140)
      COMMON /DIST/ D(70,70)
      COMMON /SAMT/ ISTOR(15,71)
      COMMON /TMTR/ ITPTOR(140)
      CALL OBTAIN(FNAME,ONAME,N)
      OPEN(3,FILE=ONAME,STATUS='NEW',FORM='FORMATTED')
      TORDIS=99999.99
      CALL NODE(N,NUM,SAM)
      CALL NBR(NUM,SAM,DIS)
      DO 7 L=1,5
         DO 8 J=1,N+1
            ITOUR(J)=ISTOR(L,J)
8        CONTINUE
         CALL MARKD()
         DO 100 C=1,N
            CALL TRPREP(C)
            R=0
120         I=1
            IT(1)=ITOUR(1)
            IT(2)=ITOUR(2)
130         CALL SELCTY(I,C,G,NOD)
            I=I+1
            IF(G.LE.(0.0001)) GOTO110
            CALL ADJTOR(I,R,NOD)
            GOTO130
110         IF(R.EQ.1)GOTO100
            CALL REVTOR(R)
            GOTO120
100      TDIS=0.0
         DO 140 J=1,N
            TDIS=TDIS+D(ITOUR(J),ITOUR(J+1))
140      CONTINUE
         IF(TDIS.GE.TORDIS)GOTO7
         TORDIS=TDIS
         DO 6 J=1,N+1
            ISTOR(L,J)=ITOUR(J)
6        CONTINUE
7     CONTINUE
```

```
      DO 28 I=1,N+1
         ITOUR(I)=ISTOR(1,I)
20    CONTINUE
      CALL TPPREP(1)
      WRITE(*,29) FNAME
      WRITE(3,29) FNAME
      WRITE(*,30) TOPDIS
      WRITE(3,30) TOPDIS
      WRITE(*,31)
      WRITE(3,31)
      WRITE(*,32)(ITOUR(I),ANAME(ITOUR(I)),I=1,N+1)
      WRITE(3,32)(ITOUR(I),ANAME(ITOUR(I)),I=1,N+1)
      CLOSE(3)
      STOP
29    FORMAT(1X,'The solution for data file ',A)
30    FORMAT(1X,'Total tour distance = ',F8.2)
31    FORMAT(1X,'The tour is as follows:')
32    FORMAT(1X,I3,5X,A)
      END
C
      SUBROUTINE OBTAIN(FNAME,ONAME,N)
        IMPLICIT CHARACTER*15 (A)
        INTEGER N
        REAL TIME
        CHARACTER*15 FNAME,ONAME
        COMMON /NAME/ ANAME(70)
        COMMON /DIST/ D(70,70)
        WRITE(*,6)
        READ(*,7) FNAME
        WRITE(*,8)
        READ(*,7) ONAME
        OPEN(4,FILE=FNAME,FORM='FORMATTED')
        READ(4,9,ERR=14) N
        TIME=.584+.62*N+.0914*N**2+.0000094*N**3
        WRITE(*,10) TIME
        WRITE(*,11)
        READ(4,12,ERR=14) ((D(I,J),J=1,N),I=1,N)
        READ(4,7,ERR=14) (ANAME(I),I=1,N)
        CLOSE(4)
      RETURN
14      WRITE(*,15)
        CLOSE(4)
      STOP
6       FORMAT(1X,'Under what file name is the distance matrix? ')
7       FORMAT(A)
8       FORMAT(1X,'File name for the results? ')
```

```
9         FORMAT(I3)
1         FORMAT(1X//1X,'It will take approximately ',F6.2,' seconds')
11        FORMAT(1X,'to solve this problem.  Please wait.'///)
12        FORMAT(F7.2)
15        FORMAT(1X,'Error upon load')
        END
C
        SUBROUTINE NODE(N,NUM,SAM)
          INTEGER N,J,K,ITEMP,NUM,SAM(15)
          COMMON /RAND/ IX,IY,IZ
          IX=15363
          IY=25467
          IZ=30179
          NUM=5+NINT(2.0*ALOG(FLOAT(N-4)))
          DO 2 J=1,NUM
3             ITEMP=NINT(RANDOM(1.0)*FLOAT(N))
              IF(ITEMP.EQ.0)GOTO3
              F=0
              DO 4 K=1,NUM
                 IF(ITEMP.EQ.SAM(K))F=1
4             CONTINUE
              IF(F.EQ.1)GOTO3
              SAM(J)=ITEMP
2         CONTINUE
        RETURN
        END
C
        SUBROUTINE MEP(NUM,SAM,DIS)
          INTEGER NUM,I,J,K,INODE,ITEMP,L,SAM(15),NODE
          REAL TEMPD,DIS(15)
          COMMON /WALK/ N,ITOUR(71)
          COMMON /DIST/ D(70,70)
          COMMON /SAMT/ ISTOR(15,71)
          COMMON /TDST/ TD(70,70)
          DO 10 L=1,NUM
          INODE=SAM(L)
          DO 20 I=1,N
             DO 20 J=1,N
                TD(I,J)=D(I,J)
20           CONTINUE
             TD(I,INODE)=9999.99
20        CONTINUE
          ISTOR(L,1)=INODE
          DIS(L)=0.0
          DO 21 I=1,N-1
             ITEMP=1
```

```
            TEMPD=TD(ISTOR(L,I),ITEMP)
            DO 22 J=2,N
               IF(TD(ISTOR(L,I),J).GE.TEMPD)GOTO22
               ITEMP=J
               TEMPD=TD(ISTOR(L,I),J)
22          CONTINUE
            ISTOR(L,I+1)=ITEMP
            DIS(L)=DIS(L)+TEMPD
            DO 23 K=1,N
               TD(K,ITEMP)=99999.99
23          CONTINUE
21       CONTINUE
         ISTOR(L,N+1)=ISTOR(L,1)
         DIS(L)=DIS(L)+D(ISTOR(L,N),ISTOR(L,1))
10       CONTINUE
205      NOEX=1
         DO 210 I=1,NUM-1
            IF(DIS(I).EQ.DIS(I+1))DIS(I+1)=99999.99
            IF(DIS(I).LE.DIS(I+1))GOTO210
            NOEX=0
            TEMPD=DIS(I+1)
            DIS(I+1)=DIS(I)
            DIS(I)=TEMPD
            DO 220 J=1,N+1
               ITEMP=ISTOR(I+1,J)
               ISTOR(I+1,J)=ISTOR(I,J)
               ISTOR(I,J)=ITEMP
220         CONTINUE
210      CONTINUE
         IF(NOEX.NE.1)GOTO205
      RETURN
      END
C
      SUBROUTINE TPPREP(C)
        INTEGER C,J,K
        COMMON /WALK/ N,ITOUR(71)
        COMMON /TMTP/ ITPTOR(140)
        J=0
        DO 40 K=1,N
           ITPTOR(K)=ITOUR(K)
           ITPTOR(K+N)=ITOUR(K)
           IF(ITPTOR(K).EQ.C) J=K
40      CONTINUE
        DO 41 K=1,N
           ITOUR(K)=ITPTOR(J+K-1)
41      CONTINUE
```

```
      ITOUR(N+1)=ITOUR(1)
      RETURN
      END
C
      SUBROUTINE MARKD()
        INTEGER J,K
        COMMON /WALK/ N,ITOUR(71)
        COMMON /TDST/ TD(70,70)
        DO 52 J=1,N
           DO 51 K=1,N
              TD(J,K)=-1.0
51         CONTINUE
           TD(J,J)=1.0
52      CONTINUE
        DO 52 J=1,N-1
           TD(ITOUR(J),ITOUR(J+1))=1.0
           TD(ITOUR(J+1),ITOUR(J))=1.0
52      CONTINUE
        TD(ITOUR(N),ITOUR(1))=1.0
        TD(ITOUR(1),ITOUR(N))=1.0
      RETURN
      END
C
      SUBROUTINE SELCTY(I,C,G,NOD)
        INTEGER I,C,M,K,COL
        INTEGER F,STN(5),STLN(5),ND(5),NOD
        REAL G,TEMP,SY(5),SLA(5),STOP,TARRAY(71)
        COMMON /WALK/ N,ITOUR(71)
        COMMON /TDST/ TD(70,70)
        COMMON /INVT/ IT(140)
        COMMON /DIST/ D(70,70)
        DO 65 J=1,N
           TARRAY(J)=TD(ITOUR(2),J)
65      CONTINUE
        DO 62 K=1,5
           F=0
           TEMP=9999.99
           DO 61 J=3,N
              IF(D(ITOUR(2),ITOUR(J)).GE.TEMP)GOTO61
              IF(TARRAY(ITOUR(J)).GT.0.0)GOTO61
              TEMP=D(ITOUR(2),ITOUR(J))
              F=J
61         CONTINUE
           IF(F.NE.0)GOTO64
           M=M-1
           M=5
```

```
            GOTO64
64          TARRAY(ITOUR(F))=1.0
            SY(K)=TEMP
            SLA(F)=D(ITOUR(1),ITOUR(2))-D(ITOUR(F-1),ITOUR(1))
            STN(K)=ITOUR(F)
            STLA(K)=ITOUR(F-1)
            SLA(K)=SLA(K)+D(ITOUR(F),ITOUR(F-1))-SY(K)
            ND(K)=F-1
            M=K
60        CONTINUE
          STOP=0.0
          IF(M.EQ.0)STOP=9999.99
          TEMP=SLA(1)
          COL=1
          DO 63 K=1,M
             IF(SLA(K).LE.TEMP)GOTO63
             TEMP=SLA(K)
             COL=K
63        CONTINUE
          C=SLA(COL)+STOP
          NOD=ND(COL)
          IT(2*I+1)=STN(COL)
          IT(2*I+2)=STLA(COL)
        RETURN
        END
C
        SUBROUTINE ADJTOR(I,R,NOD)
          INTEGER I,J,R,NOD
          COMMON /WALK/ N,ITOUR(71)
          COMMON /TDST/ TD(70,70)
          COMMON /INVT/ IT(140)
          COMMON /TMTR/ ITPTOR(140)
          DO 81 J=2,NOD
             ITPTOR(NOD+2-J)=ITOUR(J)
81        CONTINUE
          DO 82 J=2,NOD
             ITOUR(J)=ITPTOR(J)
82        CONTINUE
          TD(IT(2*I-2),IT(2*I-1))=1.0
          TD(IT(2*I-1),IT(2*I-2))=1.0
          TD(IT(2*I-1),IT(2*I))=1.0
          TD(IT(2*I),IT(2*I-1))=1.0
          TD(IT(2*I),IT(1))=1.0
          R=0
        RETURN
        END
```

```
C
      SUBROUTINE REVTOP(R)
        INTEGER R
        COMMON /WALK/ N,ITOUP(21)
        COMMON /TMIP/ ITPTOP(110)
        DO 92 J=1,N+1
           ITPTOP(N+2-J)=ITOUP(J)
92      CONTINUE
        DO 91 J=1,N+1
           ITOUP(J)=ITPTOP(J)
91      CONTINUE
        R=1
      RETURN
      END
C
      FUNCTION RANDOM(A)
      REAL TEMP
      COMMON /RAND/ IX,IY,IZ
      IX=171*MOD(IX,177)-2*(IX/177)
      IY=172*MOD(IY,176)-2*(IY/176)
      IZ=170*MOD(IZ,178)-2*(IZ/178)
      IF(IX.LT.0)IX=IX+30269
      IF(IY.LT.0)IY=IY+30307
      IF(IZ.LT.0)IZ=IZ+30323
      TEMP=FLOAT(IX)/30269.0+FLOAT(IY)/30307.0+FLOAT(IZ)/30323.0
      RANDOM=AMOD(TEMP,1.0)
      RETURN
      END
```

APPENDIX D

LISTING OF MATRIX UTILITIES PROGRAM

```
C       *** DISTANCE MATRIX UTILITIES ***
        IMPLICIT CHARACTER*15 (A)
        INTEGER IANS
        COMMON /DIST/ D(100,100)
        COMMON /NAME/ ANAME(100)
        WRITE(*,1)
        WRITE(*,2)
        WRITE(*,3)
12      WRITE(*,4)
        WRITE(*,5)
        WRITE(*,6)
        WRITE(*,7)
        WRITE(*,8)
        WRITE(*,9)
        WRITE(*,10)
        READ(*,11) IANS
        IF((IANS.LT.1).OR.(IANS.GT.6))GOTO12
        IF(IANS.EQ.1)CALL MATRIX()
        IF(IANS.EQ.2)CALL EXTRCT()
        IF(IANS.EQ.3)CALL CORECT()
        IF(IANS.EQ.4)CALL MATPRT()
        IF(IANS.EQ.5)CALL DIST()
        IF(IANS.NE.6)GOTO12
        STOP
1       FORMAT(1X,'This is a set of utilities to be used with')
2       FORMAT(1X,'the program ROUTE.RUN.  They are to help  ')
3       FORMAT(1X,'create and manipulate a distance matrix.')
4       FORMAT(//1X,'1 - create a distance matrix')
5       FORMAT(1X,'2 - extract a small matrix from a larger matrix')
6       FORMAT(1X,'3 - correct a previously stored matrix')
7       FORMAT(1X,'4 - create a hardcopy of a distance matrix')
8       FORMAT(1X,'5 - list names with their row number in matrix')
9       FORMAT(1X,'6 - exit this program')
10      FORMAT(//1X,'What is your selection? '\)
11      FORMAT(I1)
        END
C
        SUBROUTINE MATRIX()
          IMPLICIT CHARACTER*15 (A)
          REAL DIST
          INTEGER N,I,J,L
          COMMON /DIST/ D(100,100)
          COMMON /NAME/ ANAME(100)
          WRITE(*,31)
          READ(*,32) N
          WRITE(*,33)
```

```
      DO 34 I=1,N
         WRITE(*,35) I
         READ(*,36) ANAME(I)
34    CONTINUE
      WRITE(*,37) M,N
      WRITE(*,38)
      DO 39 I=1,N-1
         D(I,I)=9999.99
         DO 40 J=I+1,N
            WRITE(*,41) I,J
            READ(*,42) DIST
            D(I,J)=DIST
            DO 100 L=1,1000
100         CONTINUE
            D(J,I)=DIST
            IF(DIST.GE.0)GOTO40
            J=N
            I=N-1
40       CONTINUE
39    CONTINUE
      D(N,N)=9999.99
      CALL STORE(N)
    RETURN
31    FORMAT(//1X,'How large is the problem? '\)
32    FORMAT(BN,I3)
33    FORMAT(//1X,'Input names for homeport and aids')
35    FORMAT(1X,'Number',I3,'  Name? '\)
36    FORMAT(A)
37    FORMAT(1X,I3,' BY ',I3,' Matrix')
38    FORMAT(' Input distances from point I to point J')
41    FORMAT(  Distance from pt ',I3,' to pt ',I3,' = '\)
42    FORMAT(BN,F7.2)
    END
C
    SUBROUTINE EXTRCT()
      IMPLICIT CHARACTER*15 (A)
      INTEGER N,M,I,J,IROW(100)
      REAL SD(100,100)
      CHARACTER*15 ASYMM(100)
      COMMON /DIST/ D(100,100)
      COMMON /NAME/ ANAME(100)
      WRITE(*,10)
      WRITE(*,11)
      WRITE(*,12)
      WRITE(*,13)
      CALL OBTAIN(N)
```

```
      WRITE(*,14)
      READ(*,15)M
      DO 23 I=1,M
         WRITE(*,21) I
21       FORMAT(1X,I3,'.  Row number from main matrix: '\)
         READ(*,15)IROW(I)
         DO 22 J=1,N
            SD(I,J)=D(IROW(I),J)
22       CONTINUE
         ASMNM(I)=ANAME(IROW(I))
23    CONTINUE
      DO 23 J=1,N
         DO 24 I=1,M
            D(I,J)=SD(I,IROW(J))
24       CONTINUE
         D(J,J)=9999.99
         ANAME(J)=ASMNM(J)
23    CONTINUE
      CALL STORE('')
    RETURN
10    FORMAT(//1X,'To create a smaller matrix from your main')
11    FORMAT(1X,'distance matrix, enter the row numbers of')
12    FORMAT(1X,'main matrix pairs which you desire in the ')
13    FORMAT(1X,'smaller matrix')
14    FORMAT(//1X,'What is the size of the smaller matrix? '\)
15    FORMAT(BN,I3)
    END
C
    SUBROUTINE CORECT()
      IMPLICIT CHARACTER*15 (A)
      INTEGER N,I,J,K,IANS,IP,IC
      REAL DIST
      COMMON /DIST/ D(100,100)
      COMMON /NAME/ ANAME(100)
      CALL GRIMIN(N)
      WRITE (*,14)
      WRITE(*,15)
16    WRITE(*,17)
      WRITE(*,18)
      READ(*,10) IANS
      IF(IANS.EQ.3)GOTO25
      IF(IANS.EQ.2)GOTO25
      IF(IANS.NE.1)GOTO16
      WRITE(*,21)
      READ(*,12) IP
      WRITE(*,22)
```

```
      READ(*,12) IC
      WRITE(*,23) IR,IC,D(IR,IC)
      WRITE(*,24)
      READ(*,13) DIST
      D(IR,IC)=DIST
      DO 100 K=1,1000
100   CONTINUE
      D(IC,IR)=DIST
      GOTO16
25    WRITE(*,26)
      WRITE(*,27)
      READ(*,12) IR
      WRITE(*,28)
      READ(*,12) IC
      DO 30 I=IR,N-1
         DO 31 J=IC,N
            WRITE(*,23) I,J,D(I,J)
            WRITE(*,24)
            READ(*,13) DIST
            IF(DIST.GE.0.0)GOTO32
            I=N
            J=N
            GOTO31
32          D(I,J)=DIST
            DO 110 K=1,1000
110         CONTINUE
            D(J,I)=DIST
31       CONTINUE
         IC=I+2
         D(I,I)=9999.99
30    CONTINUE
      GOTO16
35    CALL STOPE(N)
      RETURN
12    FORMAT(BN,I3)
13    FORMAT(F7.2)
14    FORMAT(//1X,'This utility is for correcting or completing')
15    FORMAT(1X,'a previously stored data file.')
17    FORMAT(1X,'1 - Single entries      2 - Sequential entries')
18    FORMAT(1X,'3 - Exit corrections          Selection:'\)
19    FORMAT(I1)
21    FORMAT(1X,'Row number to correct '\)
22    FORMAT(1X,'Column number to correct '\)
23    FORMAT(1X,'Present distance from',I3,' to ',I3,' = ',F7.2)
24    FORMAT(1X,'New distance = '\)
26    FORMAT(1X,'To end entry of data, type a negative distance')
```

```
27       FORMAT(1X,'Starting row number '\)
28       FORMAT(1X,'Starting column number '\)
      END
C
      SUBROUTINE MATPRT()
        IMPLICIT CHARACTER*15 (A)
        INTEGER I,J,N,IS,IE,JS,JE
        COMMON /DIST/ D(100,100)
        COMMON /NAME/ ANAME(100)
        WRITE(*,3)
        WRITE(*,4)
        WRITE(*,5)
        WRITE(*,6)
        CALL OBTAIN(N)
        OPEN(3,FILE='MATPRT.DATA',STATUS='NEW',FORM='FORMATTED')
        IS=1
        IE=N
        IF(N.GT.50)IE=50
7       JS=1
        JE=9
9       DO 8 I=IS,IE
           IF(N.LT.10) JE=N
           DO 10 J=JS,JE
              WRITE(3,11) D(I,J)
10         CONTINUE
           WRITE(3,12)
8       CONTINUE
        WRITE(3,13)
        JS=JE+1
        JE=JE+9
        IF(JE.GT.N) JE=N
        IF(JS.LE.N) GOTO9
        IS=IE+1
        IE=IE+50
        IF(IE.GT.N) IE=N
        IF(IS.LE.N) GOTO7
        CLOSE(3)
        WRITE(*,14)
      RETURN
3       FORMAT(//1X,'This utility prepares a file containing the ')
4       FORMAT(1X,'distance matrix for the printer.  You may have')
5       FORMAT(1X,'to cut & paste some sections to get the matrix')
6       FORMAT(1X,'in a by n form.')
11      FORMAT(F8.2\)
12      FORMAT('  .')
13      FORMAT(' ..................................................')
```

```
14      FORMAT(' Use FORMAT command to obtain print of MATRIX.DATA')
      END
C
      SUBROUTINE LIST()
        IMPLICIT CHARACTER*15 (A)
        INTEGER N,I
        COMMON /DIST/ D(100,100)
        COMMON /NAME/ ANAME(100)
        WRITE(*,10)
        WRITE(*,11)
        WRITE(*,12)
        WRITE(*,13)
        CALL OBTAIN(N)
        OPEN(3,FILE='LIST.DATA',STATUS='NEW',FORM='FORMATTED')
        DO 14 I=1,N
           WRITE(*,15) I,ANAME(I)
           WRITE(3,15) I,ANAME(I)
14      CONTINUE
        CLOSE(3)
      RETURN
10      FORMAT(//1X,'This routine will list aid row number and')
11      FORMAT(1X,'aid name.  The list will scroll by the   ')
12      FORMAT(1X,'screen and be saved to disk under file   ')
13      FORMAT(1X,'LIST.DATA use FORMAT command for hardcopy')
15        FORMAT(1X,I3,5X,A)
      END
C
      SUBROUTINE OBTAIN(N)
        IMPLICIT CHARACTER*15 (A)
        INTEGER N,I,J
        CHARACTER*15 FNAME
        COMMON /DIST/ D(100,100)
        COMMON /NAME/ ANAME(100)
        WRITE(*,10)
        READ(*,11) FNAME
        OPEN(4,FILE=FNAME,FORM='FORMATTED')
        READ(4,12,ERR=14) N
        READ(4,13,ERR=14)((D(I,J),J=1,N),I=1,N)
        READ(4,11,ERR=14)(ANAME(I),I=1,N)
        CLOSE(4)
      RETURN
14      WRITE(*,15)
        CLOSE(4)
      STOP
10      FORMAT(//1X,'Under what file is the distance matrix? ')
11      FORMAT(A)
```

```
12      FORMAT(I3)
13      FORMAT(F7.2)
15      FORMAT(1X,'Error during load')
      END
C
      SUBROUTINE STORE(N)
        IMPLICIT CHARACTER*15 (A)
        INTEGER N,I,J
        CHARACTER*15 FNAME
        COMMON /DIST/ D(100,100)
        COMMON /NAME/ ANAME(100)
        WRITE(*,10)
        READ(*,11) FNAME
        WRITE(*,12) FNAME
        OPEN(3,FILE=FNAME,STATUS='NEW',FORM='FORMATTED')
        WRITE(3,13,ERR=16) N
        WRITE(3,14,ERR=16)((D(I,J),J=1,N),I=1,N)
        WRITE(3,11,ERR=16)(ANAME(I),I=1,N)
        CLOSE(3)
        WRITE(*,15)
        GOTO18
16      WRITE(*,17)
18      CLOSE(3)
      RETURN
10      FORMAT(//1X,'File name for matrix to be saved? '\)
11      FORMAT(A)
12      FORMAT(' The file will be saved under ',A)
13      FORMAT(I3)
14      FORMAT(F7.2)
15      FORMAT(1X,'File saved without error')
17      FORMAT(1X,'Error while saving file')
      END
```

LIST OF REFERENCES

1. Parker,R.G. and Rardin,R.L., "The Traveling Salesman Problem: An Update of Research", Naval Research Logistics Quarterly 30 (1983), p. 69-96

2. Rosenkrantz,D.J, Stearns,R.E., and Lewis,P.M., "An Analysis of Several Heuristics for the Traveling Salesman Problem", SIAM Journal of Computing ,vol. 6, no. 3 (1973), p. 563-581

3. Bellmore,M. and Nemhauser,G.L., "The Traveling Salesman Problem: A Survey", Operations Research 16 (1968), p. 538-558

4. Golden,B. et al., "Approximate Traveling Salesman Algorithms", Operations Research 28 (1980), p. 694-711

5. Held,M. and Karp,R.M., "A Dynamic Programming Approach to Sequencing Problems", SIAM 10 (1962), p.196-210

6. Little,J.D. et al., "An Algorithm for the Traveling Salesman Problem", Operations Research 11 (1963), p. 972-989

7. Held,M. and Karp,R.M., "The Traveling Salesman Problem and Minimum Spanning Trees", Operations Research 18 (1970), p. 1138-1162

8. Held,M. and Karp,R.M., "The Traveling Salesman Problem and Minimum Spanning Trees: Part II", Mathematical Programming 1 (1971), p. 6-25

9. Held,M. and Karp,R.M., "The Traveling Salesman Problem and Minimum Spanning Trees", Operations Research 18 (1970), p. 1139

10. Hansen,K.H. and Krarup,J., "Improvements of the Held-Karp Algorithm for the Symmetric Traveling Salesman Problem", Mathematical Programming 7 (1974), p. 87-96

11. Houck,D.J. and others, "The Travelling Salesman Problem as a Constrainted Shortest Path Problem: Theory and Computional Experience", OPSEARCH, Vol. 17, No. 2 & 3, p. 93-108, 1980.

12. Bazaraa,M.S. and Goode,J.J., "The Traveling Salesman Problem: A Duality Approach", Mathematical Programming 13 (1977), p. 221-237

13. Dantzig,G.,Fulkerson,R., and Johnson,S., "Solution of a large Scale Traveling Salesman Problem", Operations Research 2 (1954), p. 393-410

14. Dantzig,G.,Fulkerson,R., and Johnson,S., "On a Linear Programming, Combinatorial Approach to the Traveling" Operations Research 7 (1959), p. 59-66

15. Dantzig,G.,Fulkerson,R., and Johnson,S., "Solution of a large Scale Traveling Salesman Problem", Operations Research 2 (1954), p. 397

16. Miliotis,P., "Integer Programming Approaches tc the Traveling Salesman Problem", Mathematical Programming 10 (1976), p. 367-378

17. Miliotis,P., "Using Cutting Planes to Solve the Symmetric Travelling Salesman Problem", Mathematical Programming 15 (1978), p. 177-188

18. Grotschel,M., "On the Symmetric Travelling Salesman Problem: Solution of a 120 City Problem", Mathematical Programming Study 12 (1980), p. 61-77

19. Padberg,M.W. and Hong,S., "On the Symmetric Travelling Salesman Problem: A Computational Study", Mathematical Programming Study 12 (1980), p. 87-96

20. Croes,G.A., "A Method For Solving Traveling Salesman Problems", Operations Research 6 (1958), p. 791-812

21. Lin,S., "Computer Solutions of the Traveling Salesman Problem", Bell System Technical Journal, Vol. 44, No. 10, p. 2245-2269, 1965.

22. Lin,S. and Kernighan,B.W., "An Effective Heuristic Algorithm for the Traveling Salesman Problem", Operations Research 21 (1973), p. 498-516

23. Christofides,N. and Eilon,S., "Algorithms For Large Scale Travelling Salesman Problems", Operations Research Quarterly, Vol. 23, No. 4, p. 511-518, 1972.

24. Golden,B. et al., "Approximate Traveling Salesman Algorithms", Operations Research 28 (1980), p. 701

25. Rosenkrantz,D.J, Stearns,R.E., and Lewis,P.M., "An Analysis of Several Heuristics for the Traveling Salesman Problem", SIAM Journal of Computing ,vol. 6, no. 3 (1973), p. 577

26. Wichmann,B.A. and Hill,I.D., "An Efficient and Portable Pseudo-random Number Generator", Applied Statistics, Vol. 31, No. 2, p. 188-190, 1982.

27. Karg, R.L. and Thompson, G.L., "A Heuristic Approach to Solving Travelling Salesman Problems", Managment Science, Vol. 10, No. 2, p. 225-248, 1964.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Superintendent Attn: Library, Code 0142 Naval Postgraduate School Monterey, Califcrnia 93943	2
3.	Ccmmandant (G-PTE) U.S. Coast Guard Washington, D.C. 20590	2
4.	Ccmmandant (G-NSR) U.S. Coast Guard Washington, D.C. 20590	1
5.	Ccmmanding Officer USCGC BLACKHAW (WLB 390) Attn: Lt. Richard Lang U.S. Ccast Guard Base Yerba Buena Island San Francisco, California 94130	1
6.	Asscc. Professor G.F. Lindsay, Code 55Ls Department of Operations Research Naval Postgraduate School Monterey, Califcrnia 93943	2
7.	Asst. Professor K. Wood, Code 55Wd Department of Operations Research Naval Postgraduate School Monterey, Califcrnia 93943	1
8.	Ccmmanding Officer USCGC PENOBSCOTT BAY (WTGB 107) Attn: Lt. Jon Bechtle Governors Island New York, New York 10004	2

END

FILMED

4-85

DTIC